Implementaciones paralelas del algoritmo FastSepNMF para la identificación de endmembers en el procesamiento de imágenes hiperespectrales

---

Parallel implementations of the FastSepNMF algorithm for endmember identification in hyperspectral image processing

Aroa Ayuso Muñoz
David Savary Martínez

Trabajo de Fin de Grado en Ingeniería Informática
Facultad de Informática, Universidad Complutense de Madrid

Madrid, septiembre de 2020

Directores:

Sergio Bernabé García
Guillermo Botella Juan

Índice general

Índice de figuras

Índice de tablas

Resumen

Durante las últimas décadas, el uso de imágenes hiperespectrales se ha expandido a diversos sectores. Gracias a los avances en los sensores encargados de producir estas imágenes y al estudio de diversas técnicas para procesarlas, las imágenes hiperespectrales se han incorporado a actividades como la extracción minera, detección de objetivos en el ámbito militar o la detección y seguimiento de catástrofes ambientales. La obtención del resultado en un tiempo aceptable es clave en ciertas actividades, convirtiéndose en un objetivo principal el procesamiento de estas imágenes a tiempo real o próximo a este.

El principal uso de las imágenes hiperespectrales es el análisis de la composición de una superficie, mediante un proceso de desmezclado espectral en el que intervienen distintos algoritmos. En primer lugar se ha de identificar la cantidad de materiales presentes en la imagen. Posteriormente se extraen sus firmas espectrales, llamadas *endmembers*, para finalmente generar un mapa con la abundancia de estos *endmembers*.

Este trabajo se centra en mejorar el rendimiento del algoritmo FastSepNMF, "Fast and robust recursive algorithm for separable NMF", cuya implementación permite la extracción de las firmas espectrales de los *endmembers* a partir de píxeles puros de una imagen. A lo largo de este proyecto se han implementado distintas versiones del algoritmo haciendo uso de varios lenguajes de programación paralela: OpenMP, OpenCL y CUDA. Tras obtener estas versiones, se ha estudiado su rendimiento al compararlas con una implementación secuencial del algoritmo. Además, se ha analizado la eficiencia de las versiones paralelas en las dos plataformas presentes (CPUs multicore y GPUs) en un sistema HPC heterogéneo.

Palabras clave

Imágenes hiperespectrales, FastSepNMF, OpenCL, OpenMP, CUDA, procesamiento paralelo, optimización, aceleración, firma espectral, endmember.

Abstract

During the last decades, hyperspectral imaging has been introduced into several fields. Thanks to the development of more capable hyperspectral sensors and the study of various processing techniques, hyperspectral images have been incorporated into activities such as mining extraction, target detection for military purposes and monitoring of environmental disasters. Some of these activities have time constraints for delivering a response, making real-time processing one of the most important goals to achieve.

The main use of hyperspectral imaging is analyzing the composition of a surface through a spectral unmixing process that involves several steps. Firstly, the amount of materials present in the image must be identified. Then, the spectral signatures (the *endmembers*) of those materials are extracted. Finally, using the *endmembers*, it is possible to map the distribution of materials present in the studied surface.

This paper focuses on improving the performance of the FastSepNMF algorithm (Fast and robust recursive algorithm for separable NMF), which allows extracting the spectral signatures of the *endmembers* by selecting pure pixels in an image. Throughout this project, different versions of the algorithm have been implemented using three parallel programming languages: OpenMP, OpenCL and CUDA. The performance of these implementations has been studied and compared with a sequential implementation of FastSepNMF. Additionally, the efficiency of the parallel versions when executed on the two platforms (multicore CPUs and GPUs) present in an heterogeneous HPC system has been analyzed.

Keywords

Hyperspectral images, FastSepNMF, OpenCL, OpenMP, CUDA, parallel processing, optimization, acceleration, spectral signature, endmember.

Capítulo 1

Introducción

1.1. Motivación

La visión y entendimiento del universo evolucionan constantemente debido a las nuevas técnicas que se han desarrollado hasta el momento para observarlo. Actualmente nos encontramos estudiando campos que requieren de tecnologías muy complejas que exigen cada vez mas precisión y eficiencia para apreciar detalles que no podemos conocer de otro modo. Dentro de las técnicas de teledetección se encuentran las imágenes hiperespectrales, que recientemente se han usado para estudiar la superficie terrestre con sensores situados en aeronaves y plataformas espaciales [3] sin necesidad de realizar experimentos sobre el terreno.

Las imágenes hiperespectrales son una evolución de las imágenes digitales que conocemos. Estas últimas están tomadas en el espectro electromagnético de la luz visible y formadas por tres bandas espectrales (rojo, verde y azul). Por el contrario, las imágenes hiperespectrales se toman en un espectro electromagnético con mayor amplitud, incluyendo el infrarrojo y el ultravioleta, permitiendo observar cientos de bandas cada una correspondiente a una longitud de onda. La información que nos aportan estas imágenes nos permite diferenciar

entre otras cosas, los distintos materiales que están presentes en un área a través de la resolución del problema principal de este tipo de imágenes, la mezcla espectral.

Son muchas las actividades que se benefician del uso de las imágenes hiperespectrales. Entre ellas podemos encontrar aplicaciones mineras, que permiten distinguir los distintos tipos de minerales presentes en una zona, aplicaciones militares para la detección de objetivos o aplicaciones medioambientales. En esta última nos permiten estudiar los cambios y deterioros en ecosistemas, los impactos del cambio climático y el calentamiento global o la detección y seguimiento de incendios y desastres ambientales. Por otro lado, se ha incorporado el uso de este tipo de imágenes en distintas industrias, como la alimenticia, para la detección y clasificación de alimentos con ciertas características [4].

Uno de los problemas a los que está sujeta esta tecnología es la cantidad de datos que hay que procesar. Estos datos acaban siendo enviados para el procesamiento en tierra y una parte muy significativa son almacenados en bases de datos sin ser nunca tratados. Algunas de las actividades anteriormente mencionadas requieren resultados en un tiempo real o muy próximo a este, por lo que un objetivo claro es el procesamiento de estas imágenes en las mismas plataformas donde se sitúan los sensores. Esto reduciría el tiempo y el coste de procesamiento, creando la necesidad de mejorar los algoritmos que existen actualmente [5].

Para resolver este problema, se propone paralelizar una de las etapas del procesamiento de las imágenes para poder reducir el tiempo de los cálculos. Para ello se usarán paradigmas de programación paralela como CUDA, OpenCL y OpenMP y se probarán sobre CPUs multicore y GPUs (Unidades de Procesamiento Gráfico) que los soporten. Nuestro trabajo se centrará en aprovechar estas tecnologías permitiéndonos obtener los resultados en el menor tiempo posible y ayudar a conseguir un procesamiento en tiempo real.

1.2. Objetivos

El objetivo general de este proyecto es el estudio y aplicación de distintos lenguajes para paralelizar y acelerar el algoritmo FastSepNMF (Fast and Robust Recursive Algorithms for Separable Nonnegative Matrix Factorization) [6]. Con este algoritmo se puede resolver una de las tres fases necesarias para estimar la abundancia de los materiales presentes en una imagen hiperespectral.

Se aplicará la programación paralela con memoria compartida usando OpenMP y la programación heterogénea usando OpenCL y CUDA al algoritmo FastSepNMF. Posteriormente se pasará a hacer una comparativa de los resultados obtenidos con cada versión sobre las distintas plataformas presentes en la máquina de pruebas.

El objetivo general de este proyecto se lleva a cabo a lo largo de esta memoria a través de la ejecución de varios objetivos específicos, los cuales se muestran a continuación:

- Estudio y comprensión de los conceptos del campo a tratar. Definición de las imágenes hiperespectrales, concepto *endmember* y el proceso de desmezclado espectral.
- Estudio del algoritmo FastSepNMF.
- Traducción del algoritmo al lenguaje C. Optimización de la versión secuencial en C favoreciendo la vectorización automática por el compilador.
- Estudio de los paradigmas de programación paralela con los que se va a acelerar el algoritmo.
- Implementación de las distintas versiones paralelas del algoritmo.
- Obtención y comparativa de resultados entre las distintas versiones implementadas del algoritmo, en términos de rendimiento, calidad y eficiencia.

1.3. Plan de trabajo

En la Figura 1.1 encontramos un diagrama de Gantt con la planificación de las distintas actividades a realizar. En morado se encuentra el flujo normal que se ha seguido para la realización de las tareas. En color naranja se encuentran las tareas para las que se habían dado por finalizadas pero han sido retomadas al no cumplir con los objetivos o requerir modificaciones al adquirir mayor conocimiento sobre ellas. En ambos casos se diferencian dos patrones, a rayas si en el intervalo de tiempo correspondiente estuvo en desarrollo la tarea y liso para indicar que se finalizó en algún momento del intervalo de tiempo indicado.

1.4. Organización de esta memoria

Habiendo definido los hitos en los que se ha organizado el objetivo general, nos ocupa ahora especificar la estructura en la que se va a desarrollar la memoria, compuesta por distintos capítulos los cuales se detallan brevemente a continuación:

- **Análisis hiperespectral**: En este apartado se tratarán los principales conceptos que se han de conocer sobre el campo en el que se desarrolla el trabajo. Para ello se definirán las imágenes hiperespectrales y el problema de la mezcla espectral, el cual aborda el proceso de extracción de *endmembers*. Por último se expondrá la necesidad de paralelizar este proceso usando distintas plataformas para acelerar esta técnica de teledetección.

- **Implementación**: Tras conocer el contexto en el que se desarrolla el proyecto, en este capítulo se profundizará en el algoritmo FastSepNMF. Se definirá su funcionalidad, además de las distintas transformaciones y optimizaciones que se han aplicado durante su traducción a C. Se explicarán los paradigmas de la programación paralela OpenMP, OpenCL y CUDA utilizados para las implementaciones aceleradas del

Figura 1.1: *Planificación y contingencias del plan de trabajo.*

algoritmo optimizado. También se detallará el proceso de implementación de estas versiones aceleradas. El código de las versiones desarrolladas se encuentra alojado en GitHub[1].

- **Resultados**: Habiendo finalizado el desarrollo, se procederá a obtener los resultados. En este capítulo se estudiarán distintos aspectos como el rendimiento y el consumo de las distintas versiones implementadas así como la calidad de los resultados del algoritmo. Para cada resultado no solo influye la imagen y la versión del algoritmo utilizado, sino también la plataforma hardware sobre la que se ejecutará. Complementando a lo anterior, se describirán las características de las imágenes y las plataformas usadas durante las pruebas.

- **Conclusiones y trabajo futuro**: Por último se expondrán las conclusiones extraídas de los resultados, determinando si se ha cumplido el objetivo de este proyecto. Además, se plantearán posibles líneas de investigación a desarrollar en el futuro con relación al trabajo presentado.

[1] https://github.com/savary1/FastSepNMF_parallelization

Capítulo 2

Análisis hiperespectral

2.1. Imágenes hiperespectrales

Una imagen hiperespectral es el resultado de medir la reflectancia (cantidad de radiación electromagnética reflejada) por una superficie utilizando un sensor hiperespectral. Estos sensores analizan una región concreta del espectro electromagnético, midiendo la intensidad de la radiación que se refleja en distintas longitudes de onda para cada uno de los píxeles que forman la imagen [3]. Por lo tanto, una imagen hiperespectral está formada por tantas listas de reflectancias como píxeles tenga. Estas listas son las llamadas "firmas espectrales" de los píxeles y contienen un valor para cada una de las longitudes de onda que mide el sensor. La firma espectral de un píxel está relacionada con la capacidad de los materiales que lo componen de absorber y reflejar radiación electromagnética en longitudes de onda concretas. Esto hace posible determinar los materiales presentes en un píxel a partir de su firma espectral.

Como representa la Figura 2.1, los datos de una imagen hiperespectral se pueden organizar en forma de cubo, como si se apilasen las fotografías de cada banda espectral. En este cubo dos ejes representan las posiciones de los píxeles, y el tercero, las diferentes bandas

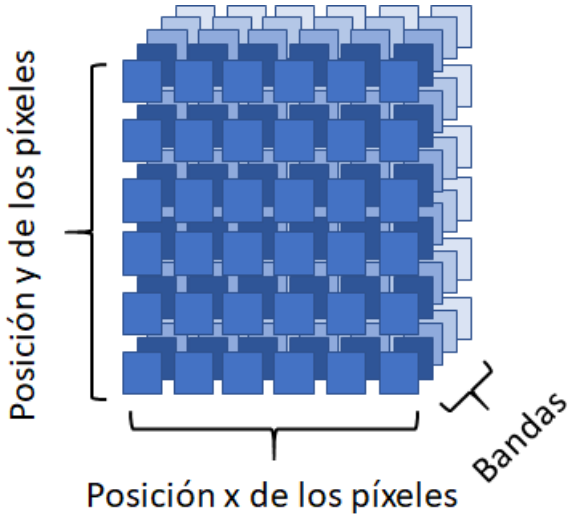


Figura 2.1: *Imagen hiperespectral representada como un cubo de datos.*

estudiadas. La firma espectral de cada píxel estaría entonces representada por los valores de este tercer eje [7].

Las imágenes hiperespectrales tienen dos características muy deseables que las diferencian de otros tipos de imágenes. La primera es que abarcan una franja amplia del espectro electromagnético sin limitarse a la luz visible (la mayoría de los sensores hiperespectrales abarcan también partes de la región infrarroja), por lo que contienen una gran cantidad de información [8]. La segunda es que tienen una gran resolución espectral. Muchos sensores multiespectrales toman medidas en bandas muy anchas, además separadas por franjas en las que no se realiza ninguna medición. Los sensores hiperespectrales en cambio, realizan las mediciones en bandas contiguas de anchura más reducida. La gran cantidad de pequeñas bandas contiguas producen imágenes con mucha más resolución espectral, permitiendo captar detalles importantes para diferenciar materiales con firmas espectrales similares [9]. Esta mayor resolución espectral se puede aprovechar usando técnicas específicas para las imágenes hiperespectrales [10].

2.2. El problema de la mezcla espectral

Aunque en la superficie de un objeto los materiales estén mezclados a escala microscópica, un sensor hiperespectral solamente es capaz de tomar una única firma espectral por cada píxel que forma la imagen. Sin embargo, es posible determinar los diferentes materiales que están presentes en un píxel y la proporción en la que están mezclados mediante un proceso de *spectral unmixing* (desmezclado espectral) [4].

En el proceso de desmezclado se llama *endmember* a la firma espectral que tendría un píxel puro formado únicamente por uno de los materiales de la imagen. En la Figura 2.2 se muestra una imagen hiperespectral formada por píxeles puros y píxeles formados por más de un *endmember*. Como se explica en [7], el concepto de píxel puro es relativo al problema que se quiere estudiar. Si se tuviese una imagen hiperespectral de una zona boscosa, se podrían suponer solamente dos *endmembers*, uno para representar el suelo descubierto y otro para representar la superficie cubierta por vegetación. Esto sería suficiente para realizar un estudio del crecimiento del bosque. Si en cambio se quisiera realizar un estudio sobre el desarrollo del bosque, es posible que tener un *endmember* por cada tipo de planta diferente sea lo más útil. De esta forma se detectaría cuánto y a qué zonas se ha expandido cada una de las distintas plantas si se toman varias imágenes a lo largo del tiempo. En otro tipo de estudio, la cantidad de clorofila presente en la escena se podría utilizar para detectar algún tipo de anomalía. En ese caso sería interesante tener un *endmember* que represente la clorofila. El número de *endmembers* depende en gran medida del problema que se quiere solucionar.

Una vez determinado el número de *endmembers*, la tarea del desmezclado espectral consiste en separar la firma espectral de cada píxel en los *endmembers* que la forman y la abundancia de esos *endmembers* [4]. Para obtener las abundancias, el método más simple y utilizado es el modelo de mezcla lineal. Este modelo asume que los píxeles se pueden describir como la combinación lineal de los *endmembers* que lo componen. En esta combinación lineal,

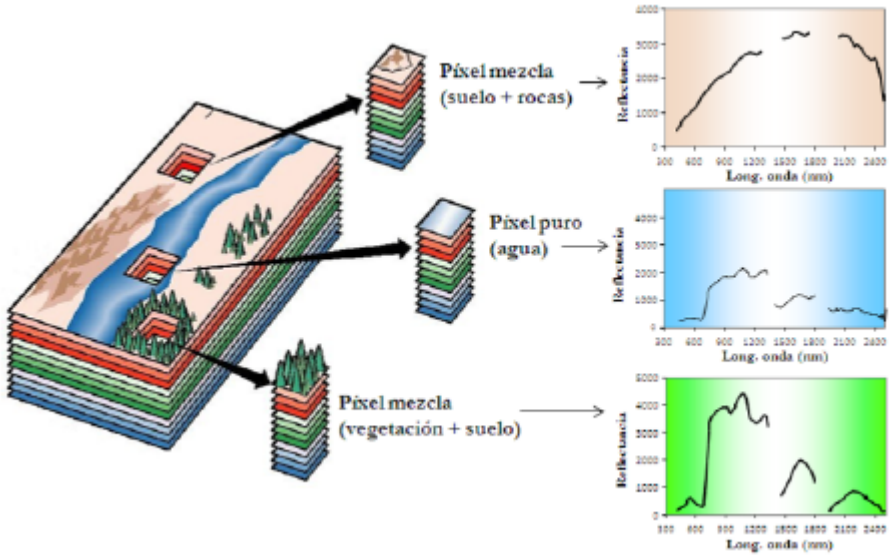


Figura 2.2: *En una imagen hiperespectral se pueden encontrar píxeles puros y píxeles formados por más de un endmember [1].*

el peso de la firma espectral de cada *endmember* es el porcentaje de la superficie que ocupa en el píxel. El modelo de mezcla lineal no es una representación exacta de la realidad, ya que la luz interactúa de manera diferente con cada material, e incluso puede interactuar con varios materiales antes de llegar al sensor. Existen también modelos no lineales, que tienen en cuenta varios de estos efectos, a costa de ser más complejos. La Figura 2.3 muestra las diferencias entre el modelo lineal y el modelo no lineal. La mayoría de los algoritmos de desmezclado se basan en el modelo lineal, que representa el comportamiento de la luz de forma suficientemente precisa en la mayoría de los casos y no requiere estudios previos de los materiales [7].

El modelo de mezcla lineal permite reducir el proceso de desmezclado espectral a una ecuación matricial, si representamos la imagen hiperespectral como una matriz $\mathbf{M}$ en lugar de un cubo. La matriz $\mathbf{M}$ tiene entonces un tamaño $b \times n$, donde b es el número de bandas espectrales y n es el número de píxeles de la imagen. El elemento m_{ij} de la matriz $\mathbf{M}$ representa la reflectancia en la banda i del píxel j. Según la definición del modelo de mezcla

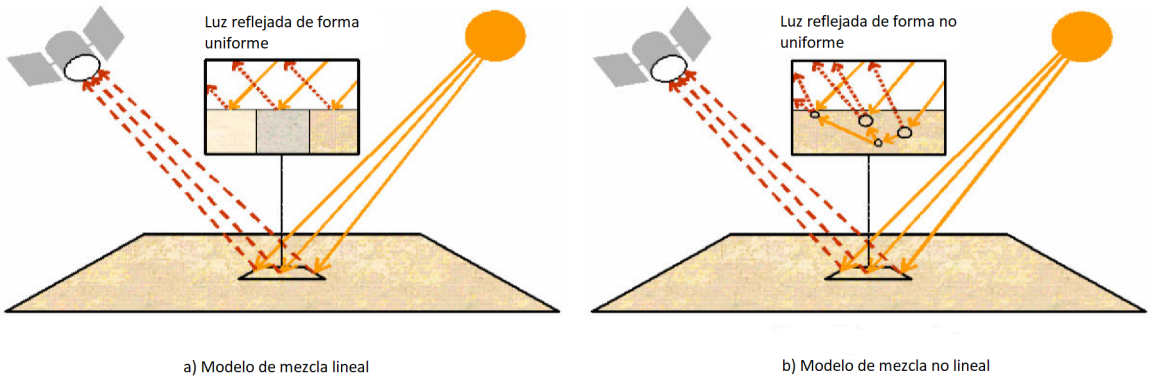


Figura 2.3: *Representación del funcionamiento de los modelos lineales y no lineales de mezcla [2].*

lineal, la reflectancia medida en un elemento m_{ij} se puede escribir como:

$$m_{ij} = \sum_{k=1}^{r} w_{ik} h_{kj} \tag{2.1}$$

donde r es el número de *endmembers*, w_{ik} es la reflectancia en la banda i del *endmember* k y h_{kj} es la abundancia del *endmember* k para el píxel j. Resolver este cálculo para todas las bandas de todos los píxeles es equivalente a resolver la ecuación $\mathbf{M} = \mathbf{WH}$ [6], donde $\mathbf{W}$ es la matriz de *endmembers* de tamaño $b \times r$ y $\mathbf{H}$ es la matriz de abundancias de tamaño $r \times n$. En este caso, la matriz $\mathbf{W}$ contendría las firmas espectrales de los *endmembers* en las columnas, y en la matriz $\mathbf{H}$ cada columna contiene las abundancias de los *endmembers* de un píxel. Existen también variaciones de este problema matricial [4] y otros algoritmos diferentes para resolver el desmezclado espectral, pero la mayoría de ellos necesitarán igualmente la matriz $\mathbf{W}$ de *endmembers*. La Figura 2.4 ilustra la solución al problema de la mezcla espectral aplicando el modelo de mezcla lineal.

Un posible método con el que obtener los *endmembers* de una imagen para poder resolver la mezcla espectral es buscarlos en una biblioteca espectral [11]. Este procedimiento no solo es

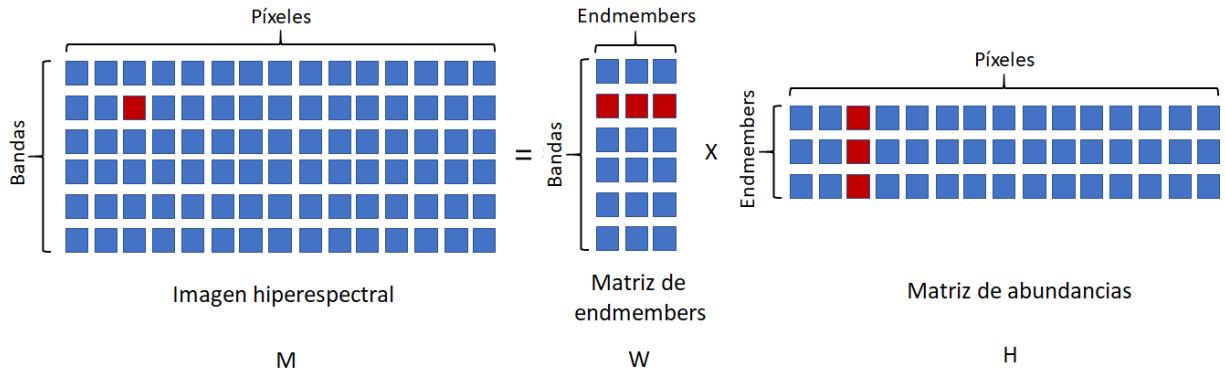


Figura 2.4: *El píxel de la imagen* $\boldsymbol{M}$ *se puede expresar como la multiplicación de la segunda fila de* $\boldsymbol{W}$ *y la tercera columna de* $\boldsymbol{H}$*. Para todos los píxeles, esto se simplifica en* $\boldsymbol{M} = \boldsymbol{WH}$*.*

costoso por el tamaño de las bibliotecas, sino también por la necesidad de adaptar la imagen en el caso de no haber sido tomada bajo las mismas condiciones que los datos obtenidos previamente. La alternativa es usar un algoritmo de extracción, que utiliza la propia imagen hiperespectral para extraer los *endmembers* necesarios al calcular las abundancias.

El algoritmo FastSepNMF [6] con el que se ha tratado en este trabajo es un algoritmo de extracción de *endmembers*. Para construir la matriz **W** de forma eficiente, este algoritmo supone que existe al menos un píxel puro para cada uno de los *endmembers*. De esta forma, al poder seleccionar píxeles de la imagen como *endmembers*, el problema se puede resolver en tiempo polinomial. Otros algoritmos que son capaces de separar **M** en el producto **WH** usando la misma técnica que FastSepNMF tienen grandes desventajas, como poca tolerancia al ruido en la imagen, un gran coste computacional o requieren de parámetros que hay que configurar aparte del número de *endmembers* [6].

2.3. Necesidad de paralelización

El principal problema de las imágenes hiperespectrales es su gran tamaño. Una de las imágenes usadas en este trabajo tiene un tamaño de 350×350 píxeles y 188 bandas, y por lo tanto, está compuesta por más de 23 millones de valores. Algunas imágenes más grandes

pueden llegan a ocupar varios cientos de *megabytes*. Estas cantidades de información necesitan mucho espacio de almacenamiento, ancho de banda para ser transmitidas y capacidad de cómputo para ser procesadas [12].

El problema se agrava cuando las tecnologías de teledetección remota se aplican a nuevas áreas. Actualmente las imágenes hiperespectrales tienen aplicaciones médicas [13] y se usan en la industria para controles de calidad en la fabricación de objetos y comida [4], además de las tradicionales aplicaciones de observación terrestre desde el aire o desde el espacio. Según se sigan encontrando nuevos usos, el flujo de información que tendrá que ser procesada irá creciendo, creando la necesidad de encontrar un modo eficiente de hacerlo.

Para solucionar los problemas causados por el tamaño de las imágenes hiperespectrales, se pueden usar varias técnicas como algoritmos de compresión [14] o descartar los datos menos útiles, acelerando los tiempos de transmisión y reduciendo el espacio de almacenamiento. El tiempo necesario para procesar las imágenes se puede reducir mediante la programación paralela [15] [16], tradicionalmente usando supercomputadores o una red de máquinas que trabajan para resolver un mismo problema. En estos últimos años, la programación paralela también se ha aplicado a las GPUs y a CPUs modernas.

Por un lado, las GPUs han visto aumentado su rendimiento en gran medida principalmente gracias a la expansión del mercado de los videojuegos y al procesado intensivo de gráficos. Por otro lado, debido al estancamiento en la reducción del tamaño de los transistores que las componen, las CPUs han comenzado a aumentar de forma exponencial el número de núcleos que incluyen y a implementar instrucciones vectoriales más potentes en los últimos años [17]. Las GPUs, las instrucciones vectoriales y las CPUs multinúcleo se pueden utilizar para acelerar en gran medida las operaciones matriciales y vectoriales en las que se basan muchos de los algoritmos utilizados para procesar imágenes hiperespectrales.

Capítulo 3

Implementación

3.1. Algoritmo FastSepNMF optimizado en serie

El algoritmo FastSepNMF [6] es un algoritmo de factorización de matrices no negativas, que separa una matriz en el producto de otras dos matrices, todas ellas sin elementos negativos. Tiene la particularidad de generar una de las dos matrices del producto a partir de columnas de la matriz original. Aplicado al procesamiento de imágenes hiperespectrales, es capaz de extraer las firmas espectrales de los *endmembers* a partir de píxeles puros en la imagen.

Se ha partido del código MATLAB que Nicolas Gillis, coautor del algoritmo junto a Stephen A. Vavasis[1], ha publicado en su página web[2] en la Sección "Fast and robust recursive algorithm for separable NMF". Este código de MATLAB requiere una imagen hiperespectral $\mathbf{M}$ de tamaño $b \times n$ (bandas×píxeles) y un número de *endmembers* r. También permite indicar si $\mathbf{M}$ necesita un proceso de normalizado para que sus columnas sumen 1. Del código de MATLAB se ha extraído el siguiente pseudocódigo (ver Algoritmo 1):

[1] https://uwaterloo.ca/scholar/vavasis
[2] https://sites.google.com/site/nicolasgillis/code

Algorithm 1 Algoritmo FastSepNMF.

```
b = numBands(M)
n = numPixels(M)

if normalize == 1 then
    for i = 0 to n-1 do
        D[i,i] = 1/(∑_{k=0}^{b-1} M[k,i] + 1e-16) {D es una matriz diagonal, cada valor i,i se compone con el píxel i a partir de todas su bandas}
    end for
    M = M * D
end if

for i = 0 to n-1 do
    normM[i] = ∑_{k=0}^{b-1} M[k,i]^2 {normM contiene los píxeles de M comprimiendo todas sus bandas}
end for
normM1 = normM
nM = max(normM)

i = 0
while i < r  &  max(normM)/nM > 1e-9 do
    a = max(normM)
    normMAux = (a - normM)/a

    numP = 0
    for j = 0 to n-1 do
        if normMAux[j] ≤ 1e-6 then
            smallP[numP] = j {smallP contiene los índices de normMAux que contengan valores menores que 1e-6}
            numP = numP + 1
        end if
    end for

    maxV = -1
    for j = 0 to numP-1 do
        if normM1[smallP[j]] > maxV then
            endm = smallP[j] {endm es la posición del mayor endmember hasta el momento}
            maxV = normM1[smallP[j]] {maxV es el mayor endmember hasta el momento}
        end if
    end for
    posE[i] = endm
    U[:,i] = M[:,endm] {Se añade a la matriz U la firma espectral del endmember extraído}

    for j = 0 to i-1 do
```

41: U[:,i] = U[:,i] - U[:,j] * (U[:,j]^t * U[:,i]) {Se actualiza la columna añadida de U a partir de las columnas anteriormente calculadas}
42: **end for**
43:
44: normF = $\sum_{k=0}^{b-1} U[k,i]^2$
45: normF = $\sqrt{normF}$
46: U[:,i] = U[:,i] / normF
47:
48: v = U[:,i] {Se forma v a partir de la última columna de U}
49: **for** j = i-1 to 0 **do**
50: v = v - (v^t * U[:,j]) * U[:,j]
51: **end for**
52: {Se actualiza normM a partir del vector v}
53: fvAux = v^t * M
54: **for** j = 0 to n-1 **do**
55: fvAux[j] = fvAux[j]^2
56: **end for**
57: normM = normM - fvAux
58:
59: i = i + 1
60: **end while**

donde max() es una función que devuelve el valor máximo de un vector y **posE** es el vector que contiene las posiciones de los píxeles seleccionados como *endmembers*.

El algoritmo consta de una iniciación, donde se normaliza la matriz **M** si es necesario, haciendo que la suma de los valores de cada píxel sumen 1 (líneas 4-8), y donde se comprimen las columnas de **M** en un vector **normM** de n posiciones (línea 12). Después, se obtienen los *endmembers* ejecutando un bucle de forma iterativa. En esta parte iterativa primero se selecciona la posición que contiene el nuevo *endmember* operando con el valor más alto de **normM** (líneas 19-37). Una vez seleccionado el *endmember*, se añade su lista de reflectancias de la imagen **M** a la matriz **U** y se obtiene a partir de ella un vector **v** (líneas 40-51). Finalmente, se actualiza la matriz **normM** con el vector **v** (líneas 53-57) para calcular el siguiente *endmember*. El proceso se detiene cuando se han encontrado los r *endmembers* o cuando el valor máximo de **normM** es demasiado pequeño para continuar.

3.2. Traducción a C y optimización de la versión secuencial

Tras estudiar el algoritmo en MATLAB, se comenzó la traducción al lenguaje C. La versión secuencial de FastSepNMF en C es la que se ha utilizado como base para las versiones aceleradas con OpenMP, OpenCL y CUDA, ya que las tres tecnologías tienen APIs disponibles en este lenguaje. Se ha optimizado esta versión todo lo que ha sido posible, de forma que al comparar los resultados de las versiones aceleradas con la versión secuencial, el aumento de rendimiento sea sobre una implementación eficiente.

La primera versión funcional fue una traducción exacta del pseudocódigo del algoritmo al lenguaje C. Esta primera versión no era muy eficiente, ya que el código de MATLAB describe a muy alto nivel las operaciones matemáticas que forman el algoritmo utilizando muchas funciones y matrices auxiliares. En C muchos de los pasos se pueden simplificar, facilitando la paralelización para las versiones posteriores y evitando cálculos innecesarios. Por esa razón, la versión secuencial final no es una implementación literal del pseudocódigo.

Por ejemplo, en las líneas 5 a 8 del Algoritmo 1 se utiliza una matriz auxiliar $\mathbf{D}$ de tamaño $n \times n$ durante el proceso de normalización de la imagen. $\mathbf{D}$ es una matriz diagonal cuyos valores se construyen a partir de una columna (píxel) de la matriz $\mathbf{M}$. En la versión secuencial final, la matriz $\mathbf{D}$ se ha reducido a tipo variable en la que se realizan los sumatorios dentro de un bucle que procesa cada píxel de forma independiente. Esto permite eliminar dependencias entre las iteraciones de los bucles necesarios durante la normalización, además de evitar la costosa multiplicación $\mathbf{M} = \mathbf{M} \times \mathbf{D}$.

Otro cambio significativo ha sido la simplificación del proceso de selección de los *endmembers* en las líneas 20 a 36 del Algoritmo 1. En el código original de MATLAB se buscan todas las posiciones de **normM** que cumplan la Ecuación 3.1 para después seleccionar el mayor de todos los valores presentes en esas posiciones. En cambio, en la versión secuencial

final, se han combinado estos dos pasos en un solo bucle que selecciona el *endmember*, comprobando que las posiciones del vector **normM** cumplan las dos condiciones (ser menor a 1e-6 y ser el mayor valor que lo cumpla hasta el momento). Esta simplificación del código ha facilitado la implementación de otras versiones posteriores.

$$\frac{a - normM_i}{a} \leq 1e^{-6} \tag{3.1}$$

Hay también otros cambios menores en todas las etapas del algoritmo. Muchos de estos cambios tienen como objetivo permitir que el compilador ICC de Intel vectorice de forma eficiente todos los bucles que implementan las operaciones con matrices y vectores, reduciendo el número de instrucciones que se ejecutan[3]. Al traducir el pseudocódigo a C, las matrices que almacenan píxeles tenían tantas columnas como píxeles y tantas filas como bandas y quedan almacenadas en memoria como una lista de filas. Por esta razón, los bucles que realizan las operaciones de sumatorios y multiplicaciones de columnas presentes en el algoritmo accedían a posiciones de memoria no consecutivas en cada iteración, separadas por tantos píxeles como la matriz contenga. Estos accesos a posiciones de memoria no adyacentes tienen como consecuencia una gran penalización en el tiempo que tarda en ejecutarse cada bucle.

Para solucionar este problema, se han traspuesto todas las matrices, de forma que ahora tienen tantas columnas como bandas y tantas filas como píxeles. También se han modificado todos los bucles que interactúan con ellas de forma que a cada iteración los datos utilizados se almacenan de forma contigua en memoria como se muestra en la Figura 3.1. Todos estos cambios reducen el tiempo de ejecución y permiten aprovechar las instrucciones vectoriales existentes en el hardware que ejecuta el algoritmo.

[3] https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/vectorization/automatic-vectorization.html

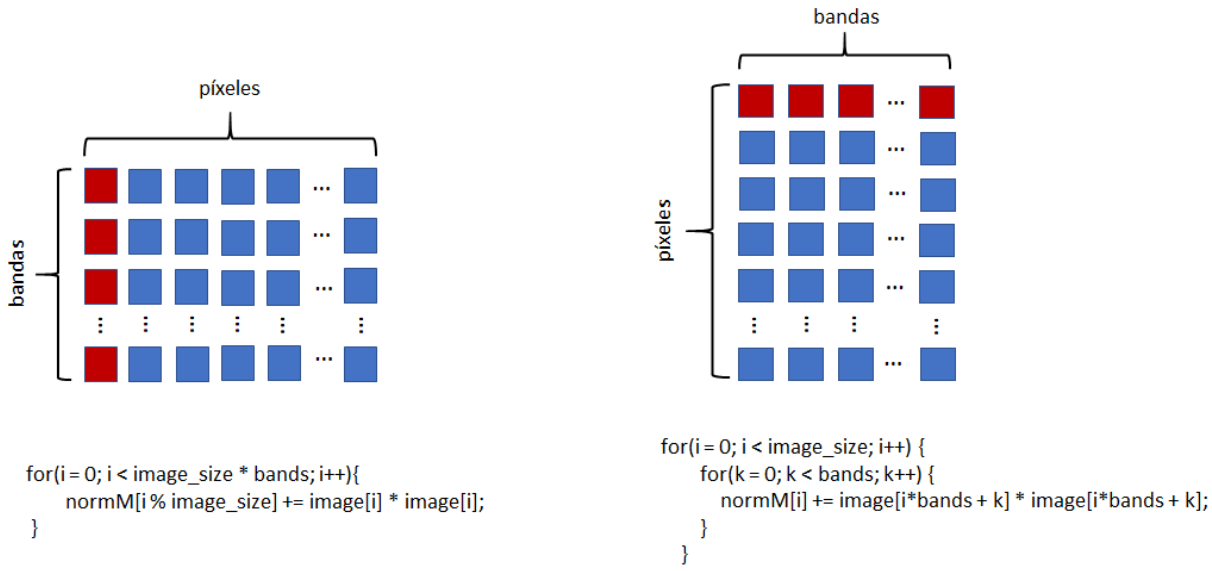


Figura 3.1: *A la izquierda, implementación de la línea 6 del Algoritmo 1 con accesos a memoria no adyacentes. A la derecha, misma línea implementada con accesos a memoria adyacentes.*

Por último, la implementación secuencial final del algoritmo y todas las que se basan en ella utilizan decimales de precisión simple al contrario que el código de MATLAB, que utiliza decimales de precisión doble. Se ha comprobado que al usar decimales de precisión simple con el tipo *float* de C en lugar de los de doble precisión con el tipo *double* se acelera la ejecución de manera notable. La desventaja de este tipo *float* es la perdida de precisión en los cálculos. Debido a ello, algunos de los píxeles que el algoritmo (operando con *floats*) señala como *endmember* no son exactamente los mismos, aunque se encuentran posicionados muy cerca a los que se calculan utilizando *doubles*. En la práctica esto no debería de ser un problema ya que los píxeles cercanos suelen tener firmas espectrales iguales o muy similares.

3.3. Paradigmas de programación paralela

Al analizar el porcentaje del tiempo que emplea cada paso del algoritmo en la versión secuencial tras la realización de un *profiling*, se observa que la normalización de la imagen, en caso de necesitarse, supone entre un 7 % y un 9 % del tiempo total, y que las líneas 53 a 57 del Algoritmo 1 suponen entre un 85 % y un 91 % del tiempo total. Estos porcentajes varían debido a que la normalización de la imagen se realiza una sola vez, mientras que las líneas 53 a 57 se ejecutan cada vez que se extrae un *endmember*. El porcentaje de tiempo que se consume procesando las líneas 53 a 57 será por tanto mayor cuantos más *endmembers* se extraigan de la imagen. Por este motivo, ha sido en estas dos partes donde más esfuerzo se ha invertido para reducir el tiempo de ejecución, desarrollado diferentes versiones de FastSepNMF para comparar la mejora de rendimiento y la eficiencia obtenida. En esta sección se revisan las características y las diferencias que existen entre las tecnologías utilizadas para la implementación de estas versiones: OpenMP, OpenCL y CUDA.

3.3.1. OpenMP

A pesar de los esfuerzos de la industria, es cada vez más complicado diseñar CPUs más potentes. Debido a esto, los ordenadores personales actuales se han especializado en la ejecución paralela de tareas para aumentar su rendimiento mediante el uso de chips con varias CPUs. Los supercomputadores a su vez, se han vuelto masivamente paralelos al fabricarse con un elevado número de procesadores con capacidad de trabajar de forma conjunta.

Una de las posibilidades para poder programar aplicaciones en sistemas de multiprocesamiento simétrico, o lo que es lo mismo, sistemas con más de una unidad de procesamiento que comparten una memoria principal es a través de una interfaz de programación de aplicaciones (API) conocida con el nombre de OpenMP. Dicha API fue creada para poder aprovechar

de forma sencilla la potencia de los computadores paralelos, de forma que no solo sean útiles ejecutando varias tareas al mismo tiempo, sino que también puedan acelerar la ejecución de una sola aplicación. En un principio se implementó en compiladores de Fortran y más tarde, en compiladores de C y C++ [18].

Se basa en el uso de directivas que el programador incluye en el código para indicar al compilador cómo tiene que realizar la paralelización [19]. Estas directivas toman la forma de *pragmas* en C y C++ y son ignoradas si el compilador no dispone de soporte para OpenMP. La directiva *parallel* indica una región paralela dentro de la cual se encuentran las instrucciones que se ejecutan de forma paralela. Al principio de una región paralela se crean varios hilos que se reparten el trabajo y que se destruyen cuando todos hayan alcanzado el final de la región. En la ejecución de un programa se van alterando regiones paralelas con regiones secuenciales, en las que solo hay un hilo de ejecución. Hay también directivas que permiten repartir el trabajo entre los hilos creados. En C y C++ la directiva *for* se utiliza para repartir las iteraciones de un bucle *for* entre todos los hilos que existen en la región paralela.

Aunque el uso de directivas simplifique en gran medida la paralelización de código secuencial, esto no significa que todo el código secuencial pueda ser paralelizado mediante OpenMP. En el caso del procesamiento de datos, la paralelización de los bucles que lo permitan suele ser muy efectiva.

3.3.2. OpenCL

La tendencia de paralelización de los computadores no solo se limita a las CPUs. Los ordenadores actuales incluyen una gran cantidad de chips dedicados a acelerar distintos tipos de procesos y muchos de ellos se basan en el procesamiento paralelo de datos y tareas. En la última década, se ha popularizado el uso de procesadores auxiliares para acelerar la ejecución de programas, destacando las GPUs [20]. Como se ha comentado en la Sección 2.3, las

GPUs han evolucionado enormemente y aunque en un principio se dedicaban exclusivamente a procesar gráficos, han evolucionado para favorecer la ejecución de código de propósito general al descubrirse su potencial en la aceleración de tareas que permiten la paralelización (GPGPU, General Purpose GPU) [21]. La responsabilidad de mejorar el rendimiento de un programa utilizando coprocesadores recae sobre el programador y requiere de gran esfuerzo si no se dispone de herramientas que faciliten el reparto de trabajo.

OpenCL surge de la cooperación entre varias compañías dedicadas a la fabricación de ordenadores y procesadores. Su objetivo era crear una tecnología multiplataforma que permitiese a un dispositivo delegar la ejecución de ciertas partes de un programa a otros procesadores heterogéneos como FPGAs, CPUs multicore, GPUs o DSPs capaces de ejecutarlas con mayor rendimiento y eficiencia. Debía, además, permitir la ejecución de un mismo código paralelo en todos los aceleradores que tengan soporte para OpenCL, sin que el programador tenga que preocuparse de crear distintas versiones para cada uno de ellos [22].

En OpenCL, se llama *host* al dispositivo que ejecuta un programa con la ayuda de otros coprocesadores o aceleradores. Se llama *device* a cada dispositivo que el *host* puede utilizar para ejecutar partes de un programa y se llama *kernel* a cada una de las partes de código que ejecuta un *device*. Aunque los diferentes *devices* que soportan OpenCL tengan características muy diferentes, todos tienen que ser capaces de ejecutar un mismo *kernel* sin necesidad de modificarlo. Esto se consigue estandarizando la comunicación entre el *host* y el *device*, el modelo de ejecución, el modelo de memoria y el lenguaje en el que se escriben los *kernels* (OpenCL C).

El modelo de ejecución describe cómo un *device* tiene que ejecutar un *kernel*, de forma que el programador pueda esperar un determinado comportamiento. En un programa, el *host* puede lanzar un número de hilos determinados por el programador. Cada uno de esos hilos, llamados *work-item*, ejecuta una instancia del *kernel*. Los *work-items* pueden comunicarse entre ellos mediante diversos mecanismos y están divididos en grupos, los *work-group*. Se

garantiza que los *work-items* de un mismo *work-group* puedan sicronizarse entre ellos ya que su ejecución es concurrente.

El modelo de memoria describe cómo un *kernel* puede interactuar con la memoria de un *device* (ver Figura 3.2). Para que las diferentes arquitecturas de los *devices* se puedan adaptar a un mismo modelo de memoria, existen 3 regiones de memoria con diferentes características:

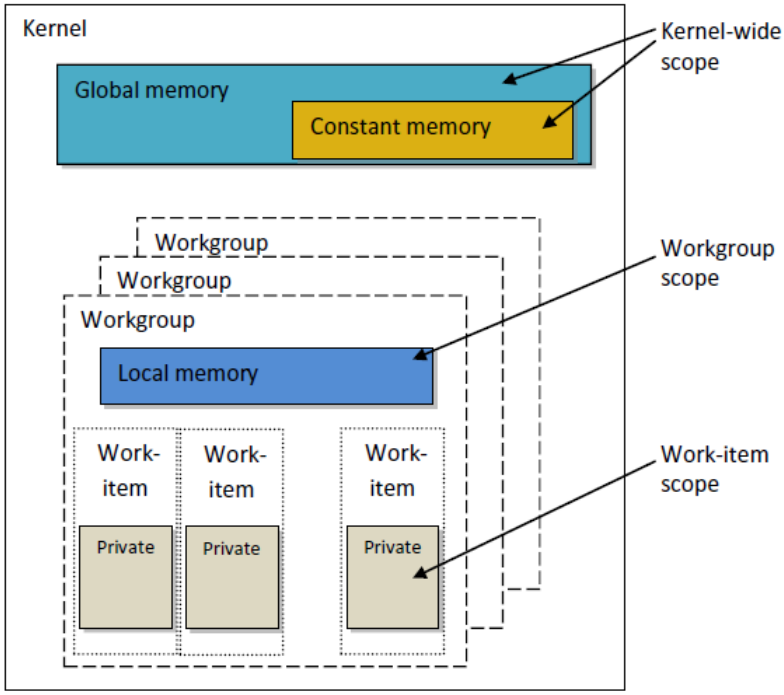


Figura 3.2: *Jerarquía memorias en OpenCL*[4].

- Memoria global: Es la memoria principal del *device*. Cualquier *work-item* puede interactuar con datos alojados en esta memoria. Se puede leer y escribir desde el *host*, permitiendo el intercambio de información entre el *host* y el *device*.

- Memoria local: Los *work-items* de un mismo *work-group* comparten la memoria local. Permite a los *devices* mapearla a hardware más rápido y escaso que el que aloja la

[4] https://www.mql5.com/en/articles/407

memoria global, permitiendo lecturas y escrituras más rápidas. Con esta memoria, los *work-items* de un mismo *work-group* pueden sincronizarse e intercambiar información de manera más eficiente.

- Memoria privada: Es única para cada *work-item*. Es aquí donde se pueden almacenar las variables locales que se necesiten.

3.3.3. CUDA

Como ya se ha mencionado antes, en las últimas décadas se ha incrementado la computación sobre tarjetas gráficas, es decir, el uso de GPUs para computación de propósito general. En el caso de Nvidia, CUDA (Arquitectura Unificada de Dispositivos de Cómputo) fue la arquitectura que desarrollaron para llevar a cabo este propósito. En 2006 tras el lanzamiento de una de sus tarjetas gráficas, que usaba esta arquitectura, publicaron la primera versión del compilador del lenguaje CUDA C, convirtiéndose en el primer lenguaje específicamente diseñado por una empresa de tarjetas gráficas para la computación de carácter general en sus chips. Además, provocó que la programación GPGPU adquiriera mayor popularidad, al ser accesible y sencilla de usar por cualquier programador. Esto provocó que por primera vez, en 2011, la supercomputadora más potente del mundo en ese momento estuviese formada por GPUs de Nvidia basadas en CUDA [23].

El propósito de este lenguaje es idéntico al de OpenCL, con la diferencia de que CUDA es exclusivo de las GPUs de Nvidia, mientras que el anterior es multiplataforma. Como en OpenCL, cuando se habla de *host* se hace referencia a la CPU y *device* hará referencia a la GPU, siendo las funciones asignadas a cada dispositivo las mismas. La arquitectura de CUDA se basa en hilos que ejecutan instancias del *kernel* de manera concurrente. Los hilos se organizan en bloques de entre una y tres dimensiones, que a su vez se agrupan formando el *grid* el cual también puede tener entre una y tres dimensiones. Además, internamente los

hilos de un bloque se dividen en *warps*, siendo esta la unidad de ejecución más pequeña. Todos los hilos de un *warp*, formado por 32 hilos, ejecutan la misma instrucción simultáneamente. El número de hilos y de bloques son definidos por el desarrollador, siempre que no se sobrepase el número de hilos por bloque permitidos por el *device* seleccionado.

Para una mayor eficiencia en el tratamiento de los datos en el *device*, CUDA dispone de una jerarquía de memorias muy similar a OpenCL, representada en la Figura 3.3. Existen cuatro memorias distintas con las que cada hilo puede interactuar de manera diferente:

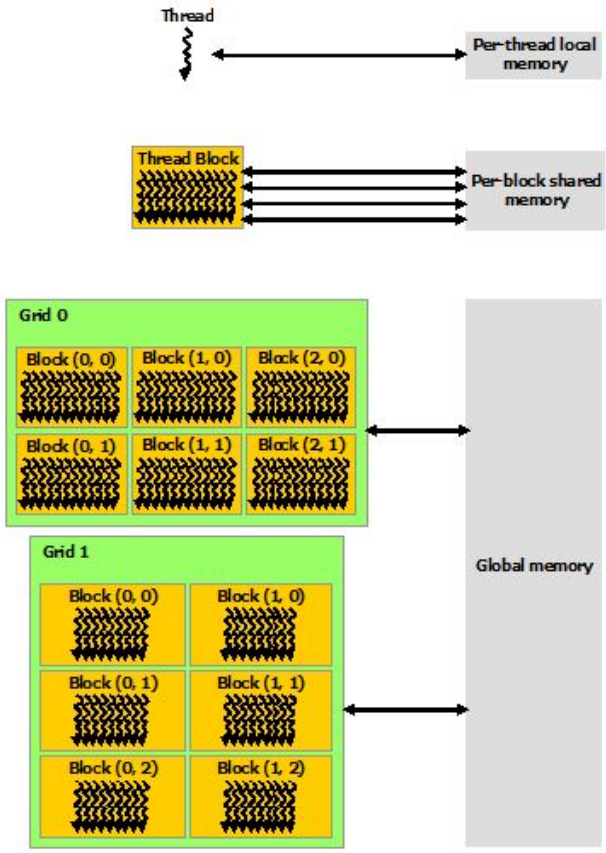


Figura 3.3: *Jerarquía memorias en CUDA*[5].

- Memoria local: Cada hilo tiene acceso a su memoria local en la que almacena sus variables privadas.

[5] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#memory-hierarchy

- Memoria compartida: Todos los hilos de un bloque pueden acceder a una misma memoria compartida, permitiendo intercambiar información rápidamente.
- Memoria global: Todos los hilos que se ejecutan en el *device* tienen acceso a esta memoria, aunque el acceso de lectura y escritura es más lento que el de la memoria compartida.
- Memoria constante: A esta memoria también puede acceder cualquier hilo, aunque únicamente para leer valores. Un kernel no puede modificar su contenido.

3.4. Implementación del algoritmo FastSepNMF usando OpenMP

Los cambios realizados a la versión secuencial del algoritmo también tienen como objetivo permitir la paralelización de las partes más costosas mediante directivas OpenMP. Esto se ha conseguido eliminando las dependencias de datos entre las iteraciones de los bucles que se han paralelizado, de forma que se puedan procesar en regiones paralelas usando tantos hilos como se desee.

En la versión OpenMP, la actualización de **normM** que es realizada entre las líneas 53 y 57 del Algoritmo 1 ocurre en una región paralela. Recordemos que esta es la parte que consume más de un 90 % del tiempo total de ejecución en algunos casos. El resultado final ha sido el Algoritmo 2, en el que cada iteración del bucle exterior actualiza una posición de la matriz **normM**. En la directiva *shared* se ha indicado las variables que deben ser privadas a cada hilo, incluyendo la variable **faux** además de los contadores de los bucles. Las variables privadas, además de estar protegidas a los cambios que realizan otros hilos, no necesitan actualizar sus cambios a la memoria compartida. Al utilizar la variable auxiliar **faux** de forma privada, se evita un gran número de costosas actualizaciones de **normM** en

la memoria compartida.

Algorithm 2 Actualización de normM (líneas 53 a 57 del Algoritmo 1) implementada con OpenMP.

```
#pragma omp parallel shared(normM, v, image, image_size, bands) private(j, k, faux)
{
        #pragma omp for schedule (static)
        for(j = 0; j < image_size; j++) {
                faux = 0;
                for(k = 0; k < bands; k++) {
                        faux += v[k] * image[j*bands + k];
                }
                normM[j] -= faux * faux;
        }
}
```

El proceso de normalizado de la imagen y el cálculo inicial de **normM** también se ha acelerado con dos regiones paralelas como se muestra en Algoritmo 3. En el caso de que sea necesario normalizar la imagen, se ejecuta la primera de las dos regiones paralelas, y en caso contrario, es la segunda región la que se ejecuta. En ambas regiones paralelas se han minimizado las actualizaciones de la memoria compartida dentro de lo posible, utilizando variables privadas en lugar de vectores ubicados en memoria compartida, como sucedía en el código original.

Se ha probado a utilizar diferentes rutinas o *schedulers* en la directiva *for* para repartir las iteraciones entre los distintos hilos, pero no se aprecia una diferencia clara de rendimiento entre ellos. *Static* es el *scheduler* por defecto, y es con el que se han realizado todas las pruebas. Con el *scheduler static*, el total de iteraciones de los bucles que se paralelizan se dividen en tantos grupos de iteraciones consecutivas como hilos se hayan creado. Cada hilo ejecuta uno de estos grupos y, por lo tanto, el trabajo se reparte de forma equitativa entre todos ellos.

Algorithm 3 Normalización de la imagen y cálculo inicial de **normM** implementado con OpenMP.

```
if (normalize == 1) {
        #pragma omp parallel shared(image, normM, normM1, image_size, bands)
        private(i, j, row, faux)
        {
                #pragma omp for schedule (static)
                for (i = 0; i < image_size ; i++) {
                        row = i*bands;
                        faux = 0;
                        for(j = 0; j < bands; j++) {
                                faux += image[row + j];
                        }
                        faux = 1.0/(faux + 1.0e-16);
                        for(j = 0; j < bands; j++) {
                                image[row + j] = image[row + j] * faux;
                        }
                        faux = 0;
                        for(j = 0; j < bands; j++) {
                                faux += image[row + j] * image[row + j];
                        }
                        normM[i] = faux;
                        normM1[i] = faux;
                }
        }
}else{
        #pragma omp parallel shared(normM, normM1, image, image_size, bands)
        private(i, k, faux)
        {
                #pragma omp for schedule (static)
                for(i = 0; i < image_size; i++) {
                        faux = 0;
                        for(k = 0; k < bands; k++) {
                                faux += image[i*bands + k] * image[i*bands + k];
                        }
                        normM[i] = faux;
                        normM1[i] = faux;
                }
        }
}
```

3.5. Implementación del algoritmo FastSepNMF usando OpenCL

Como ya se ha comentado en el apartado anterior, las iteraciones de los bucles paralelizados con OpenMP se pueden ejecutar de forma concurrente. En OpenCL las regiones paralelas pueden reescribirse en un *kernel* que se ejecute en tantos *work-items* como iteraciones tenga el bucle paralelizado. Además de escribir los *kernels*, se ha tenido que gestionar la comunicación entre el *host* y el *device*.

Algorithm 4 Actualización de normM (líneas 53 a 57 del Algoritmo 1) implementada con un *kernel* OpenCL.

```
__kernel void update_normM(__global float* restrict cl_image,
        __global float* restrict cl_v,__global float* restrict cl_normM,
        int bands, int image_size) {

    __local float l_v[224];

    unsigned int id = get_group_id(0) * get_local_size(0) + get_local_id(0);
    int k;

    if(get_local_size(0) < bands){
            for(k = get_local_id(0); k < bands; k += get_local_size(0)){
                    l_v[k]=cl_v[k];
            }
        }
    else{
        if(get_local_id(0) < bands){
            l_v[get_local_id(0)] = cl_v[get_local_id(0)];
        }
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    float faux = 0;
    if(id < image_size){
        for(k = 0; k < bands; k++){
            faux += l_v[k] * cl_image[k*image_size + id];
                }
        faux = faux * faux;
        cl_normM[id] -= faux;
    }
}
```

Por ejemplo, el *kernel* que implementa la región paralela mostrada en el Algoritmo 2 se ha implementado como se muestra en el Algoritmo 4. El bucle paralelizado con este kernel actualiza cada posición de **normM** usando el producto matricial del vector **v** y el vector de la firma espectral del píxel de **M** que se corresponde con esa posición de **normM**. Por tanto, el número de *work-items* necesarios para actualizar el vector **normM** completo será igual al número de píxeles que tenga la imagen hiperespectral. Aunque la posición de **normM** y la columna de **M** asignados a cada *work-item* son distintos, todos necesitan acceder al vector **v** en su totalidad. Para acelerar la lectura del vector **v** desde la memoria global del *device*, los *work-items* de un mismo *work-group* guardan una copia de este vector en su memoria local. Para asegurar que todas las posiciones del vector **v** se han copiado antes de calcular los nuevos valores de **normM**, se usa una barrera que sincroniza todos los *work-items* de un mismo *work-group* entre la copia de **v** y la actualización de **normM**.

Algorithm 5 Normalización de la imagen e inicialización de **normM** (cuando *normalize == 1* en Algoritmo 3) implementadas con un *kernel* OpenCL.

```
__kernel void normalize_img(__global float* restrict cl_image,
        __global float* restrict cl_normM, __global float* restrict cl_normM1,
        int image_size, int bands) {

    unsigned int id = get_group_id(0) * get_local_size(0) + get_local_id(0);
    int j;
    float imageNewVal, normAcum = 0, normAux = 0;

    if(id < image_size){
        for(j = 0; j < bands; j++){
            normAux += cl_image[j*image_size + id];
        }

        normAux = 1.0/(normAux + 1.0e-16);
        for(j = 0; j < bands; j++){
            imageNewVal = cl_image[j*image_size + id] * normAux;
            cl_image[j*image_size + id] = imageNewVal;
            normAcum += imageNewVal * imageNewVal;
        }
        cl_normM[id] = normAcum;
        cl_normM1[id] = normAcum;
    }
}
```

Algorithm 6 Inicialización de **normM** cuando no es necesario normalizar la imagen (*normalize != 1* en Algoritmo 3) implementada con un *kernel* OpenCL.

```
__kernel void initialize_normM(__global float* restrict cl_image,
        __global float* restrict cl_normM, __global float* restrict cl_normM1,
        int image_size, int bands) {

    unsigned int id = get_group_id(0) * get_local_size(0) + get_local_id(0);
    float normAcum = 0;
    int j;
    if(id < image_size){
        for(j = 0; j < bands; j++){
            normAcum += cl_image[j*image_size + id] * cl_image[j*image_size + id];
        }
        cl_normM[id] = normAcum;
        cl_normM1[id] = normAcum;
    }
}
```

Las dos regiones paralelas encargadas de inicializar **normM** en la versión OpenMP (Algoritmo 3) se han implementado con dos *kernels*, mostrados en los Algoritmos 5 y 6. El Algoritmo 5 se ejecuta cuando es necesario normalizar la imagen antes de calcular **normM** por primera vez. Para ello, se lanza un *work-item* por cada píxel que tenga la imagen. Cada uno de estos *work-items* normaliza el píxel que tenga asignado, de forma que la suma de los valores de cada píxel sea 1. Estos valores se actualizan en la matriz **cl_image** que contiene la imagen en el *device*. Después, el *work-item* calcula su valor asignado en **normM** a partir de los valores normalizados de su píxel, haciendo el sumatorio de todos los valores del píxel elevados al cuadrado. Al calcular por primera vez **normM**, se guarda en el *device* una copia en **normM1**. El vector **normM1** se utiliza en la selección de los nuevos *endmembers*, y a diferencia de **normM**, no se actualiza durante la ejecución del algoritmo. El Algoritmo 6 calcula **normM** y **normM1** a partir de la imagen original sin realizar el proceso de normalizado.

Además de los tres *kernels* que implementan cada una de las regiones paralelas de la versión OpenMP del algoritmo, se ha implementado la selección de los *endmembers* con dos *kernels* adicionales. Para seleccionar un *endmember*, es necesario encontrar el máximo

valor del vector **normM** y utilizarlo para seleccionar uno de los píxeles de la imagen, lo que también se traduce en encontrar el máximo entre ciertas posiciones del vector **normM1**. Este proceso se encuentra detallado en las líneas 19 a 36 del Algoritmo 1. Estos *kernels*, sin embargo, no realizan todo el trabajo de esta sección del algoritmo. Debido a la imposibilidad de sincronizar los *work-items* de distintos *work-groups*, no se devuelve un único valor máximo. Ambos *kernels* realizan un proceso de reducción que tiene como resultado un vector con tantas posiciones como *work-groups* hayan sido necesarios para ejecutarlos. Cada una de las posiciones de los vectores resultante contendrá el máximo local que cada *work-group* haya encontrado en la porción del vector original que se le haya asignado. El resultado final se obtiene en el *host*, buscando el máximo total entre los máximos locales. La Figura 3.4 explica el proceso de reducción.

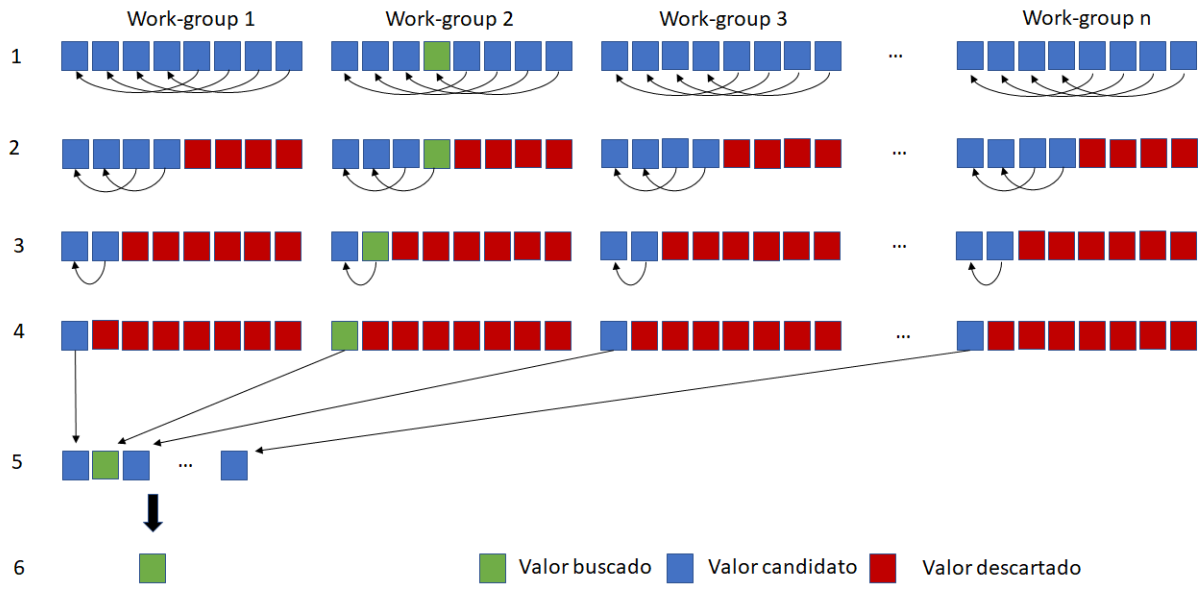


Figura 3.4: *Proceso de reducción. Los pasos 1 a 4 se realizan en el device. Cada vez que los hilos se sincronizan, la mitad de los valores de cada work-group se descartan. Al llegar al paso 4, los valores restantes de cada work-group se transfieren al host, donde se obtiene el resultado final.*

Estos *kernels* de reducción no tienen la misma eficiencia que los *kernels* que paralelizan

Algorithm 7 Implementación del primer paso de la función *max* del Algoritmo 1 en un *kernel* OpenCL para la operación de reducción. El valor final se obtiene posteriormente en el *host* a partir de los máximos locales seleccionados en el *device*.

```
__kernel void normM_reduction(__global float* restrict cl_normM,
        int image_size, __global float* restrict cl_red_result){

    int l_size = get_local_size(0);
    unsigned int id = (get_group_id(0) * 2) * get_local_size(0) + get_local_id(0);
    unsigned int l_id = get_local_id(0);
    __local float l_red[l_size];

    if(id < image_size){
        l_red[l_id] = cl_normM[id];
    }
    else{
        l_red[l_id] = -1000;
    }
    if(id + l_size < image_size){
        l_red[l_id + l_size] = cl_normM[id + l_size];
    }
    else{
        l_red[l_id + l_size] = -1000;
    }

    barrier(CLK_LOCAL_MEM_FENCE);
    for(int i = l_size; i > 0; i >>= 1) {
        if(l_id < i) {
            if(l_red[l_id] < l_red[l_id + i]){
                l_red[l_id] = l_red[l_id + i];
            }
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    if(l_id == 0) {
        cl_red_result[get_group_id(0)] = l_red[0];
    }
}
```

bucles, pero con ellos se disminuye la cantidad de datos que se mueven entre el *host* y el *device*. Si esta parte del algoritmo se calculase en el *host*, sería necesario copiar el vector **normM** completo de la memoria del *device* al *host* tantas veces como *endmembers* se extraigan. Estos *kernels* son capaces de encontrar el máximo local del *work-group* en tiempo logarítmico, ya que descartan la mitad de los valores en cada iteración hasta que solo quede uno. Como los *work-items* necesitan intercambiar información entre ellos, también se hace uso de la memoria local en estos *kernels*. El Algoritmo 7 muestra la implementación de uno de los *kernels* de reducción.

Con respecto a la gestión de la memoria del *device*, se ha puesto especial cuidado en evitar la copia de datos cuando el *device* comparte memoria con el *host*. Este es el caso cuando se utiliza el propio procesador *multicore* del *host* como *device*, o cuando el procesador del *host* contiene una GPU integrada. En estos casos se puede utilizar la opción *CL_MEM_USE_HOST_PTR* cuando se necesite inicializar una región de memoria al reservarla en el *device* con valores almacenados en el *host*. De forma similar, la función *clEnqueueMapBuffer* permite obtener un puntero en el que leer y escribir valores en la memoria global del *device*, y la función *clEnqueueUnmapMemObject* indica al *device* que se he terminado de leer y escribir en uno de los punteros obtenidos con *clEnqueueMapBuffer*. Estos tres mecanismos permiten la transmisión instantánea de información entre el *host* y el *device* sin hacer copias cuando comparten la misma memoria.

3.6. Implementación del algoritmo FastSepNMF usando CUDA

Debido a la similitud del modelo de programación entre OpenCL y CUDA, la estructura del código de estas dos versiones ha resultado ser muy parecida, aunque las funciones encargadas de la comunicación con el *device* sean muy diferentes. Se ha seguido la misma es-

trategia de crear tres *kernels* que implementan las regiones paralelas de la versión OpenMP. Dos de estos *kernels* se encargan de la parte de iniciación del algoritmo, realizando el cálculo de **normM**. Uno realiza este cálculo tras normalizar la imagen, mientras que el otro se ejecuta cuando el proceso de normalización no es necesario. El tercer *kernel* implementa la actualización del vector **normM**. Estos tres *kernels* tienen la misma estructura interna que sus versiones en OpenCL (Algoritmos 5, 6 y 4), incluso haciendo uso de la memoria compartida para agilizar la lectura del vector **v** en el tercer *kernel*.

Además de estos *kernels*, se ha explorado el uso de *kernels* de reducción, al igual que en la implementación de la versión OpenCL. La extracción de los *endmembers* consta también de dos partes, para cada una de las cuales se ha escrito un *kernel*. La primera es la obtención del valor máximo del vector **normM**, que se usa en la segunda para encontrar el nuevo *endmember* entre ciertas posiciones del vector **normM1**. El kernel que implementa este segundo paso en CUDA se puede consultar en el Algoritmo 8. Al igual que en la versión OpenCL, el último paso para encontrar los máximos se realiza en el *host*.

Después de traducir las regiones paralelas de la versión OpenMP a *kernels* en las versiones de OpenCL y CUDA, y tras la implementación de los *kernels* de reducción, el algoritmo sigue una línea de ejecución alternando entre el *host* y el *device* seleccionado. Esta línea de ejecución, que es idéntica en ambas versiones, se muestra en el esquema presente en la Figura 3.5. En él se ilustran las funciones que se ejecutan en cada dispositivo, además del número de *kernels* implementados y las transferencias de datos que se realizan durante una ejecución del algoritmo. En este esquema, los *kernels* 1 y 2 se corresponden con las dos primeras regiones paralelas de OpenMP (Algoritmo 3), mientras que el *kernel* 5 realiza la misma función que la última región paralela (Algoritmo 2). Se puede observar también que es necesario copiar **M** al *device*, que dado su tamaño es la transferencia más costosa y supone una gran parte del tiempo de ejecución de amabas versiones.

Algorithm 8 Implementación del primer paso del proceso de selección de un *endmember* en un *kernel* CUDA. El *endmember* se escoge posteriormente en el *host* de entre todos los posibles *endmembers* seleccionados en el *device*.

```
__global__ void maxValExtract(float *normM_c, float *normM1_c, long int image_size,
        float *d_projections, int *d_index, float a){
    __shared__ int pos[2048];
    __shared__ float val[2048];

    unsigned int tid = threadIdx.x;
    unsigned int id = blockIdx.x*2 * blockDim.x + threadIdx.x;

    float faux, faux2;
    faux = ((a - normM_c[id])/a);
    faux2 = ((a - normM_c[id + blockDim.x])/a);

    if(id < image_size && faux <= 1.0e-6){
            val[tid] = normM1_c[id];
            pos[tid] = id;
    }
    else{
            val[tid] = -1;
    }
    if(id + blockDim.x < image_size && faux2 <= 1.0e-6){
            val[tid + blockDim.x] = normM1_c[id + blockDim.x];
            pos[tid + blockDim.x] = id + blockDim.x;
    }
    else{
            val[tid + blockDim.x] = -1;
    }
    __syncthreads();
    for (unsigned int s = blockDim.x; s > 0; s>>=1){
            if (tid < s){
                    if(val[tid]<=val[tid+s]){
                            val[tid] = val[tid+s];
                            pos[tid] = pos[tid+s];
                    }
            }
            __syncthreads();
    }
    d_projections[blockIdx.x]=val[0];
    d_index[blockIdx.x]=(int)pos[0];

    __syncthreads();
}
```

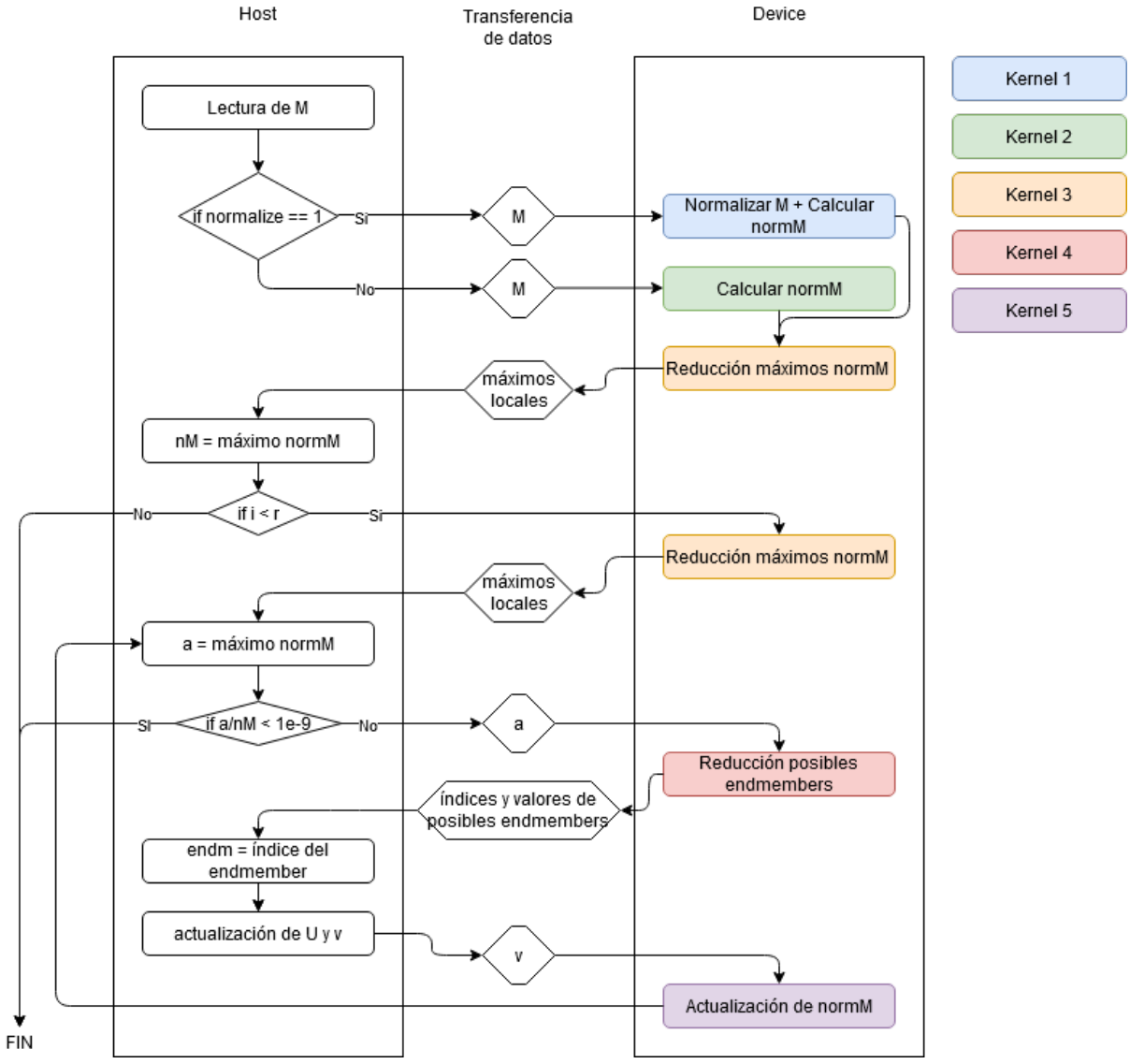


Figura 3.5: *Diagrama de ejecución de las versiones OpenCL y CUDA.*

Capítulo 4

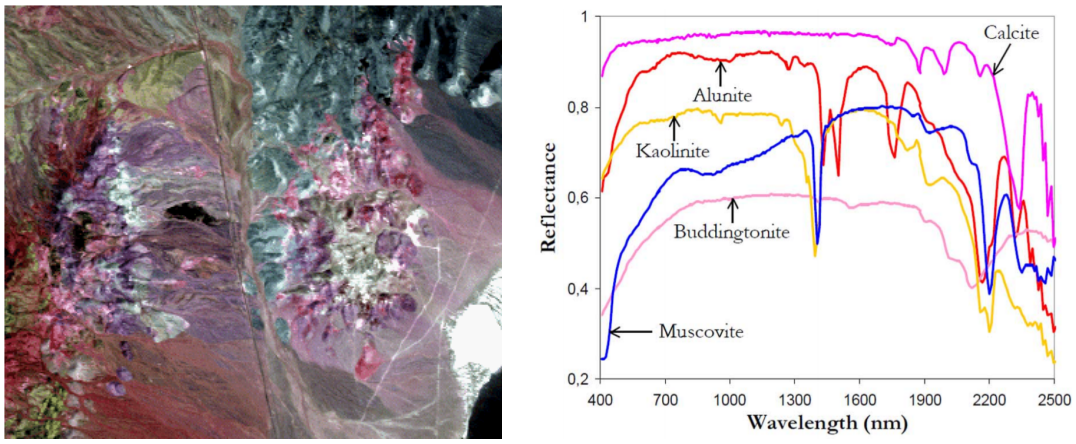
Resultados

4.1. Conjunto de datos hiperespectrales. Reales vs Sintéticos

El conjunto de datos usado está formado por imágenes hiperespectrales reales y sintéticas, sobre las que aplicando las implementaciones del algoritmo desarrollado en el transcurso de esta memoria se obtendrán los resultados pertinentes. Se han usado dos imágenes:

- La primera se trata de una de las imágenes más usadas para el problema del desmezclado hiperespectral. Fue capturada por el sensor AVIRIS (Airbone Visible/Infrared Imaging Spectrometer) de la NASA y cubre el distrito minero Cuprite, situado en Las Vegas (Nevada), en Estados Unidos. La imagen Cuprite se ha obtenido mediante mediciones sobre un terreno y sufría varias irregularidades debido a factores naturales. La imagen original está fotografiada por un espectro continuo entre los 0.4 μm y 2.5 μm de longitud de onda, obteniéndose 224 bandas espectrales. Tiene 20 metros de resolución espacial, es decir, cada píxel contiene la información de 20 metros cuadrados del terreno. Dada la extensión de la imagen y bandas con datos poco fiables se ha

extraído una parte del conjunto para su uso. La imagen final con la que se trabajará es una imagen de 350×350 píxeles en la zona con mayor número de minerales, además sin las bandas 1-4, 105-115 y 150-170 que son eliminadas debido a la absorción del agua y a la baja relación señal-ruido, quedando así 188 bandas espectrales. Se puede ver la imagen usada en la Figura 4.1(a) y las firmas espectrales de los cinco minerales principales en la Figura 4.1(b). El estudio se hará teniendo en cuenta que la zona a analizar está compuesta por 19 materiales puros (valor obtenido por estudios previos a través de algoritmos de estimación del número de *endmembers*), por lo que se definirá 19 como el número de *endmembers* a extraer. La zona observada contiene rocas que han sufrido alteraciones hidrotermales, causando mineralizaciones y alteraciones del suelo, por lo que la gran concentración de estos minerales hace de este distrito un objetivo interesante para el análisis espectral.



(a) (b)

Figura 4.1: *(a) Composición en falso color de la escena captada por el sensor hiperespectral AVIRIS sobre el distrito minero de Cuprite en Nevada. (b) Firmas espectrales de cinco de los materiales encontrados en la escena obtenidos a través de la biblioteca U.S. Geological Survey.*

- El segundo conjunto de datos utilizado es sintético. La obtención de estas imágenes parte de la utilización de algoritmos que simulan patrones naturales espaciales haciendo uso de las firmas espectrales completas de los materiales a añadir a la escena. Estas

firmas se pueden obtener de la librería espectral suministrada por el U.S.G.S. (Servicio Geológico de Estados Unidos). La imagen usada posee un tamaño de 750×650 píxeles, 224 bandas y 30 *endmembers*, es decir, 30 firmas espectrales diferentes. Esta imagen permite realizar un análisis más completo al comparar los resultados obtenidos con los de una imagen real, y así evaluar el desempeño y la escalabilidad del algoritmo. Se puede ver la imagen usada en escala de color en la Figura 4.2.

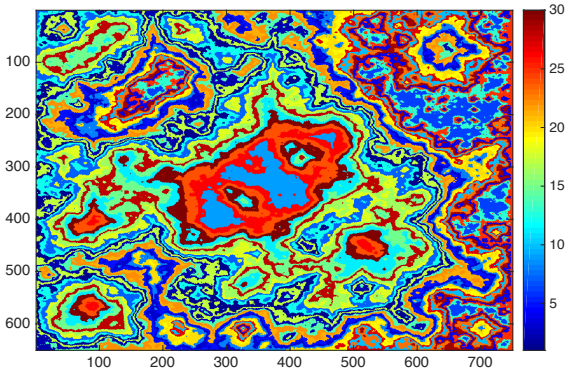


Figura 4.2: *Composición en escala de color de la escena sintética.*

4.2. Plataformas Hardware

OpenCL, CUDA y OpenMP permiten trabajar con un paralelismo a nivel de datos y de tareas haciendo que las ejecuciones sean más rápidas. Cada uno de estos lenguajes es capaz de aprovechar hardware diferente, por lo que se ha probado cada versión del algoritmo desarrollada sobre una o varias plataformas compatibles. A continuación, se describen las plataformas usadas durante las pruebas.

4.2.1. CPU Xeon

La CPU (Unidad Central de Procesamiento), además de procesar información, es la encargada de gestionar todos los recursos software y hardware conectados, administrando

el cálculo de operaciones y el intercambio de datos que se necesite en cada momento. Una de sus tareas es repartir el trabajo entre otros coprocesadores, como las tarjetas gráficas o aceleradores, permitiendo distribuir la carga de trabajo de forma eficiente.

El sistema heterogéneo utilizado contiene dos CPUs Intel Xeon CPU E5-2695 v3[1] con 14 núcleos cada una. Al disponer de tecnología SMT (Simultaneous Multithreading), estos procesadores son capaces de ejecutar 56 hilos de manera simultánea. El sistema dispone además de 64GB de memoria RAM.

La implementación del algoritmo en la versión OpenMP se beneficia de las capacidades de cómputo paralelo de estas CPUs al poder especificar la cantidad de hilos en los que repartir el trabajo. La versión implementada con OpenCL también es capaz de aprovechar la potencia de las CPUs ya que pueden adoptar el papel de *host* y *device* en un mismo programa. En cambio, la versión CUDA (y OpenCL cuando se hace uso de otro dispositivo como *device*) no permite aprovechar la gran cantidad de núcleos de los que disponen las CPUs del sistema.

Todas las versionas han sido compiladas con la versión *18.0.1* de ICC, que soporta la mayoría de funcionalidades de OpenMP *4.0* y algunas de OpenMP *5.0*[2]. Se ha utilizado la opción -O3 para obtener ejecutables con instrucciones vectoriales en todas las versiones. Además, también se ha compilado la primera traducción a C y la versión secuencial final del algoritmo con la opción -O0 para comprobar su rendimiento cuando se ejecutan sin instrucciones vectoriales. La versión de OpenCL utilizada para acelerar el algoritmo haciendo uso de los dos Intel Xeon ha sido la *1.2*.

[1] https://ark.intel.com/content/www/es/es/ark/products/81057/intel-xeon-processor-e5-2695-v3-35m-cache-2-30-ghz.html

[2] https://software.intel.com/content/www/us/en/develop/articles/intel-c-compiler-180-for-linux-release-notes-for-intel-parallel-studio-xe-2018.html#new_features

4.2.2. GPU de Nvidia

Las GPUs (Unidades de Procesamiento Gráfico) son coprocesadores creados para acelerar el procesamiento de gráficos. Aunque en las últimas décadas también se haya extendido su uso para el procesamiento de datos en los casos en los que se puedan tratar de forma paralela, su objetivo no es realizar las funciones de una CPU, sino realizar parte del trabajo de forma más eficiente. La arquitectura de las GPUs es más sencilla, están diseñadas para ejecutar una lista de instrucciones basadas en operaciones aritméticas básicas que requieren ser repetidas un número muy elevado de veces. También poseen una jerarquía de memoria optimizada para favorecer el tipo de tareas para las que están diseñadas. En el caso del algoritmo FastSepNMF, se pueden usar para acelerar las numerosas operaciones aritméticas que conlleva el procesamiento de las imágenes hiperespectrales.

El sistema heterogéneo usado cuenta con dos modelos de GPU. La primera es una Nvidia GeForce GTX 980[3], que cuenta con 2048 núcleos CUDA que funcionan a una frecuencia de 1.1 GHz y 4GB de memoria GDDR5. La segunda es una Nvidia GeForce GTX 1080[4]. Esta GPU es un modelo de una generación más reciente y potente que la del anterior y cuenta con 2560 núcleos CUDA a 1.6 GHz y 8GB de memoria GDDR5X.

Tanto la versión CUDA del algoritmo, que solamente es compatible con GPUs de Nvidia, como la versión OpenCL son capaces de aprovechar estos dos componentes. En ambas versiones, las GPUs han cumplido el papel de *device* para acelerar las partes más costosas del algoritmo, ejecutando los *kernels* descritos en el apartado de implementación. Las versiones de CUDA y OpenCL utilizadas en las pruebas con las GPUs han sido la *10.0.130* y la *1.2*, respectivamente.

[3] https://www.nvidia.com/es-es/geforce/900-series/
[4] https://www.nvidia.com/es-la/geforce/products/10series/geforce-gtx-1080/

4.3. Métricas

Existen multitud de algoritmos y versiones modificadas para la extracción de *endmembers* en el proceso de desmezclado espectral. Cada una de ellas, normalmente se centra en medir uno varios aspectos del algoritmo, como la calidad, el rendimiento o el consumo. A continuación, pasaremos a describir cada una de estas métricas.

4.3.1. Calidad

Para poder medir la eficacia del algoritmo se debe comprobar la calidad de los resultados obtenidos. Dado que el problema consiste en la extracción de *endmembers* puros, se debe saber si estos valores son correctos y en qué medida lo son. Para ello se compara la firma espectral de los *endmembers* de una imagen previamente estudiada con el *endmember* extraído por nuestro algoritmo que más se aproxime. Los *endmembers* reales de la imagen Cuprite pueden encontrarse en la biblioteca espectral de la USGS. Para realizar esta medición se usa la distancia del ángulo espectral (AE), que puede calcularse con:

$$AE(x,y) = cos^{-1}\left(\frac{\sum_{i=1}^{n} x_i y_i}{(\sum_{i=1}^{n} x_i^2)^{\frac{1}{2}} \cdot (\sum_{i=1}^{n} y_i^2)^{\frac{1}{2}}}\right) \tag{4.1}$$

Siendo $\mathbf{x}$ e $\mathbf{y}$, respectivamente, el vector con los valores en cada banda del *endmemeber* extraído y el vector con la firma espectral real del material. El resultado dará el valor del ángulo formado por los vectores. Un valor próximo a 0° indica que las firmas comparadas tendrán una similitud elevada, mientras que si es más próximo a 90°, las firmas apenas serán similares.

4.3.2. Rendimiento

Uno de los objetivos de nuestro trabajo ha sido la aceleración del algoritmo para obtener los resultados en el menor tiempo posible. A fin de evaluar la satisfacibilidad de este objetivo,

se ha de medir el rendimiento obtenido en cada versión del algoritmo, comparando la versión secuencial final vectorizada con el resto de versiones secuenciales y paralelas (OpenMP, OpenCL y CUDA). La mejora de rendimiento, o *speedup*, se calcula de la siguiente manera:

$$Speedup = \frac{T_{VersionVectorizada}}{T_{VersionAlternativa}} \tag{4.2}$$

El proceso seguido para obtener los tiempos (T) de cada versión para poder calcular la Ecuación 4.2, consiste en calcular la media de 10 ejecuciones seguidas. En cada versión se realiza esta prueba sobre la imagen Cuprite, buscando 19 *endmembers* y sobre la imagen sintética, buscando 30.

En el caso de la versión en serie y la versión OpenMP, las ejecuciones se realizaron sobre las CPUs del sistema heterogéneo. La versión OpenCL se probó con todos los *devices* disponibles, que son el propio procesador multicore de la máquina, la GPU Nvidia GTX 980 y la GPU Nvidia GTX 1080. La versión CUDA solamente se ejecutó sobre los dos modelos de GPUs incluidos en la máquina.

4.3.3. Consumo

Es posible hacer una estimación de la energía que consume la máquina de pruebas en la que se ha ejecutado las distintas versiones del algoritmo [24]. Para ello, se hizo uso del framework Pmlib [25] con arquitectura cliente/servidor capaz de medir la energía consumida por las CPUs y GPUs del sistema heterogéneo. Se puede introducir código en el algoritmo que indique al servidor cuando comienza y acaba la ejecución para obtener una serie de medidas de potencia utilizada. Obteniendo una media de la potencia consumida por cada dispositivo que haya contribuido a la ejecución del algoritmo y con la ayuda de los tiempos de ejecución, es posible calcular la energía usada y otras métricas con las que comparar la eficiencia energética de las distintas versiones del algoritmo.

Para cada versión acelerada del algoritmo se ha obtenido una medida de potencia media en Vatios (W), la cantidad de energía (E) consumida en Julios (J), y una medida de eficiencia en $Mpps/W$ (Megapíxeles por segundo / Vatio), que pone en contexto el número de píxeles (n) de la imagen que se ha procesado con la potencia desarrollada por la máquina durante la ejecución del algoritmo. Todas estas medidas son aproximaciones y se ven afectadas por el simple hecho de realizar las propias mediciones.

$$E = T \cdot W \tag{4.3}$$

$$Megapixeles = \frac{n}{1024 \cdot 1024} \tag{4.4}$$

$$Mpps/W = \frac{Megapixeles}{T \cdot W} \tag{4.5}$$

4.4. Resultados experimentales

4.4.1. Calidad

En la Tabla 4.1 se pueden observar los ángulos espectrales obtenidos de la comparación entre las firmas espectrales reales de algunos de los minerales en la escena Cuprite con los *endmembers* con mayor similitud, extraídos mediante el uso del algoritmo. Aunque estos valores no son perfectos no se puede olvidar que la presencia de alteraciones y ruido en la imagen, pueden hacer variar la firma obtenida en cada píxel. Si atendemos a los resultados obtenidos, descubrimos que los ángulos obtenidos son más cercanos a 0° que a 90°, por lo que podemos concluir que la calidad de los *endmembers* extraídos son de alta calidad, con un valor medio de 6.57° respecto a las firmas espectrales de referencia.

Alunita	Buddingtonita	Calcita	Kaolinita	Moscovita
4.81°	5.29°	5.91°	10.76°	6.10°

Tabla 4.1: *Ángulos espectrales entre los endmembers extraídos de Cuprite con FastSepNMF y las firmas espectrales de los minerales que componen la imagen.*

4.4.2. Rendimiento

En las Tablas 4.2 y 4.3 se pueden observar los tiempos de ejecución y los *speedups* ejecutando las diferentes versiones del algoritmo usando las imágenes Cuprite y Sintética. También se han realizado pruebas con el compilador GCC, pero dada la similitud entre los tiempos obtenidos, solamente se muestran los resultados del compilador ICC, que ha sido el utilizado durante el desarrollo de las distintas versiones.

Recurso	Plataforma	Tiempo (s)	Speedup
Primera versión secuencial	Intel Xeon	3.850	x0.05
Versión secuencial final	Intel Xeon	1.483	x0.14
Versión secuencial final vectorizada	Intel Xeon	0.211	x1
Versión OpenMP	Intel Xeon	0.067	x3.13
Versión OpenCL	Intel Xeon	0.082	x2.58
	GTX 980	0.030	x6.98
	GTX 1080	0.030	x6.98
Versión CUDA	GTX 980	0.022	x9.59
	GTX 1080	0.020	x10.60

Tabla 4.2: *Mejores tiempos medios de cada versión usando compilador ICC en Cuprite.*

Recurso	Plataforma	Tiempo (s)	Speedup
Primera versión secuencial	Intel Xeon	26.710	x0.05
Versión secuencial final	Intel Xeon	10.307	x0.13
Versión secuencial final vectorizada	Intel Xeon	1.384	x1
Versión OpenMP	Intel Xeon	0.449	x3.08
Versión OpenCL	Intel Xeon	0.417	x3.31
	GTX 980	0.160	x8.65
	GTX 1080	0.155	x8.93
Versión CUDA	GTX 980	0.131	x10.56
	GTX 1080	0.117	x11.77

Tabla 4.3: *Mejores tiempos medios de cada versión usando compilador ICC en la imagen Sintética.*

Se puede apreciar una gran mejora en los tiempos de ejecución entre la primera versión secuencial del algoritmo y la versión secuencial final vectorizada. La diferencia de rendimiento entre la primera versión secuencial y la versión secuencial final sin vectorización muestra que los cambios realizados al algoritmo han sido de ayuda. Por tanto, la reducción de los tiempos de ejecución no se debe solamente al uso de instrucciones vectoriales.

Aunque los *speedups* de las versiones aceleradas obtenidos en CUDA y en OpenCL con una GPU son satisfactorios, los obtenidos en la versión OpenMP y en la versión OpenCL acelerada con los procesadores Intel Xeon de la máquina no lo son tanto. Durante las pruebas se ha observado que las versiones que hacen un uso intensivo de las CPUs tienen tiempos de ejecución muy variables. La Tabla 4.4 muestra los tiempos de ejecución medios en la versión OpenMP utilizando distintos números de hilos. En ella se puede observar, por ejemplo, que el tiempo de ejecución de la imagen sintética con 28 hilos es significativamente mayor que los tiempos de ejecución de esa misma imagen con 21 y 35 hilos. Esto también se observa en menor medida en la ejecución de Cuprite acelerada con 14 hilos. El mayor *speedup* de la versión OpenMP se obtiene con cantidades bajas de hilos, debido a que aumentar el número de hilos no resulta beneficioso al procesar la imagen más grande y afecta de forma negativa a los tiempos de la imagen más pequeña. La variabilidad de los tiempos obtenidos también se observa en menor medida en la versión OpenCL con ambas imágenes.

Nº hilos	Cuprite T(s)	Sintética T(s)
7	**0.067**	0.483
14	0.077	0.549
21	0.071	**0.449**
28	0.079	0.534
35	0.093	0.457
42	0.107	0.476
49	0.126	0.496
56	0.110	0.562

Tabla 4.4: *Tiempos medios obtenidos al ejecutar la versión OpenMP con diferente número de hilos.*

A pesar de que los mayores *speedups* solamente se obtengan haciendo uso de GPUs para acelerar el algoritmo, todas las versiones excepto la primera traducción a C del algoritmo son capaces de extraer los endmembers de Cuprite en un tiempo menor que el que tarda el sensor en tomar la imagen. El sensor AVIRIS que capturó la imagen de Cuprite tarda 8.3 milisegundos en obtener los valores espectrales de 512 píxeles, lo que se traduce en que procesar la imagen Cuprite (que tiene un tamaño de 350×350 píxeles) en menos de 1.98 segundos es suficiente para conseguir un procesamiento a tiempo real [26]. Aunque el tiempo que la versión secuencial final sin instrucciones vectoriales tarda en extraer los 19 endmembers de Cuprite sea menor a esos 1.98 segundos, el margen disponible no es muy holgado. El resto de versiones, que son entre 9.4 y 100 veces más rápidas que la captura de la imagen, permiten a las demás etapas presentes en el procesamiento de las imágenes hiperespectrales disponer de más tiempo para poder generar el mapa de abundancias en tiempo real.

4.4.3. Consumo

Las medidas de consumo y eficiencia de las versiones aceleradas se pueden observar en la Tabla 4.5. No se debería comparar las eficiencias obtenidas en las dos imágenes por una misma versión, ya que aunque la medida de eficiencia tenga en cuanta el número de píxeles de las imágenes (en megapíxeles), no contempla la diferencia en la cantidad de bandas ni el número de *endmembers* extraídos.

Se puede observar que versiones aceleradas con GPUs no son solo más rápidas que las versiones aceleradas con los procesadores *multicore*, si no que también son más eficientes. Este es el resultado esperado, ya que el principal objetivo de las GPUs es mejorar el rendimiento y la eficiencia de procesos altamente paralelizables.

La eficiencia de las versiones aceleradas por GPU se ve lastrada por el alto consumo de los dos procesadores Intel Xeon, sin que estos realicen cálculos intensivos. Emparejando

una potente GPU con un procesador con menos capacidades de cálculo paralelo que los procesadores Intel Xeon E5-2695 v3 pero con mayor eficiencia, se conseguirían tiempos de ejecución similares y eficiencias mucho mayores.

Versión - Plataforma	Imagen	Tiempo (s)	Energía (J)	Potencia (W)	Mpps/W
OpenMP - CPU	Cuprite	0.067	11.78	175	0.010
	Sintética	0.449	98.09	218	0.005
OpenCL - CPU	Cuprite	0.082	17.47	214	0.007
	Sintética	0.417	96.84	232	0.005
OpenCL - GPU GTX 980	Cuprite	0.030	5.24	173	0.022
	Sintética	0.160	26.94	168	0.017
OpenCL - GPU GTX 1080	Cuprite	0.030	4.87	161	0.024
	Sintética	0.155	27.57	178	0.017
CUDA - GPU GTX 980	Cuprite	0.022	3.73	167	0.031
	Sintética	0.131	22.08	169	0.021
CUDA - GPU GTX 1080	Cuprite	0.020	3.33	167	0.035
	Sintética	0.117	20.10	171	0.023

Tabla 4.5: *Resultados de eficiencia obtenidos a partir de la potencia consumida por las distintas versiones y sus tiempos medios.*

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

Este trabajo se ha centrado en el estudio y la explotación del potencial de paralelización del algoritmo FastSepNMF. El algoritmo, basado en el modelo de mezcla lineal, es capaz de extraer *endmembers* de una imagen hiperespectral con una alta tolerancia al ruido y un coste computacional no muy elevado al asumir la presencia de píxeles puros en la imagen. Según los autores de FastSepNMF en el artículo [6] publicado a finales de 2013, ningún otro algoritmo similar posee estas ventajas.

Para realizar este estudio, se han implementado diferentes versiones de FastSepNMF usando distintos paradigmas de la programación paralela. Se ha aplicado la programación paralela con memoria compartida usando OpenMP y la programación heterogénea usando OpenCL y CUDA. Tanto la versión secuencial que se ha implementado para realizar las comparaciones como las versiones paralelas, se han optimizado para aprovechar las capacidades de un sistema HPC heterogéneo compuesto por: 2×Nvidia GPUs (una Nvidia GeForce GTX 1080 y una Nvidia GeForce GTX 980), además de 2×CPUs Intel Xeon E5-2695 v3.

Para las pruebas experimentales, el algoritmo se ha probado con dos imágenes hiperes-

pectrales distintas. La primera, Cuprite, está tomada sobre una zona de Las Vegas (Nevada), Estados Unidos, y representa un caso real muy estudiado. La segunda es una imagen sintética de gran tamaño que se ha usado para comprobar que las distintas versiones del algoritmo escalan bien con imágenes de un tamaño mayor de píxeles.

Como resultado, se ha obtenido una gran mejora de rendimiento con las versiones CUDA y OpenCL que hacen uso de la GPU para acelerar el procesamiento. Los tiempos obtenidos con OpenCL-GPU han sido casi 9 veces más rápidos que los de la versión secuencial, mientras que el aumento de rendimiento es más pronunciado en la versión de CUDA que procesa la imagen sintética casi 12 veces más veloz. En el caso de OpenCL-CPU y OpenMP, los resultados no han sido tan significativos, obteniendo tiempos en torno a 3 veces más rápidos. Todas las versiones aceleradas se ejecutan en tiempos muy inferiores al límite para obtener un procesamiento en tiempo real para la imagen Cuprite.

Las versiones aceleradas más eficientes en cuanto a consumo energético son, de nuevo, CUDA y OpenCL-GPU en este orden. En tercer y cuarto lugar se encuentran las versiones OpenMP y OpenCL-CPU. Estos son los resultados esperados, ya que las GPUs tienen una eficiencia mucho mayor que las CPUs en escenarios altamente paralelizables.

El estudio concluye que la versión CUDA es la más eficiente. No obstante, la versión OpenCL tiene eficiencias parecidas cuando se acelera con una GPU y tiene la ventaja de soportar muchas más GPUs aparte de las de Nvidia, además de CPUs *multicore* y otros tipos de aceleradores.

5.2. Trabajo futuro

Para continuar con el estudio del algoritmo FastSepNMF, se propone comprobar el comportamiento de las distintas versiones implementadas sobre hardware diferente al usado en el sistema HPC heterogéneo y estudiar su escalabilidad con imágenes reales de mayores dimen-

siones. Comparando los rendimientos obtenidos con las versiones OpenCL-CPU y OpenMP en otras configuraciones de CPUs diferentes a los dos Intel Xeon utilizados en las pruebas de este trabajo, se podría desvelar el motivo de los tiempos de ejecución tan variables que se obtienen.

De forma similar, se puede estudiar la eficiencia y rendimiento de las versiones CUDA y OpenCL-GPU cuando se ejecutan usando una GPU potente y un procesador de menor consumo que los Intel Xeon del sistema heterogéneo. Estos procesadores tienen una gran capacidad de cálculo paralelo, que no es tan necesaria en el caso de acelerar la ejecución del algoritmo mediante una GPU. Encontrar una configuración de hardware óptima es necesario antes de aplicar el algoritmo a cualquier situación del mundo real.

Por último, se podría analizar la aplicación del algoritmo a entornos controlados como los controles de calidad en el ámbito industrial. En estos escenarios, es más probable tener la seguridad de que exista al menos un píxel puro por cada *endmember* que se quiera extraer. Para estos casos sería conveniente que el algoritmo se ejecutara sobre hardware de dimensiones reducidas que se pueda transportar fácilmente junto al sensor, como una FPGA (usando la versión OpenCL) o un SoC (System on a Chip) parecido a los usados en *smartphones*.

Bibliografía

[1] Milosz Ciznicki, Krzysztof Kurowski, and Antonio Plaza. GPU implementation of JPEG2000 for hyperspectral image compression. *Proc SPIE*, 8183, 10 2011.

[2] Antonio Plaza, Gabriel Martin, Javier Plaza, Maciel Zortea, and Sergio Sanchez. *Recent developments in endmember extraction and spectral unmixing*, pages 235–267. 01 1970.

[3] Avid Roman-Gonzalez and Natalia Indira Vargas-Cuentas. Análisis de imágenes hiperespectrales. *Revista Ingenieria & Desarrollo*, 35:14–17, 2013.

[4] Antonio Plaza, Javier Plaza, Abel Paz, and Sergio Sanchez. Parallel hyperspectral image and signal processing. *IEEE Signal Processing Magazine*, 28:119–126, 01 2011.

[5] Pier Luigi Dragotti and Michael Gastpar. *Distributed Source Coding, theory algorithms and applications*, chapter 10:Distributed Compression of Hyperspectral Imagery. Academic Press, 2009.

[6] Nicolas Gillis and Stephen A. Vavasis. Fast and robust recursive algorithmsfor separable nonnegative matrix factorization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36.4 (2014):698–714, 2014.

[7] Jose M. Bioucas-Dias, Antonio Plaza, Nicolas Dobigeon, Mario Parente, Qian Du and Paul Gader. Hyperspectral unmixing overview: Geometrical, statistical, and sparse regression-based approaches. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 2012.

[8] Michael T. Eismann. *Hyperspectral remote sensing.* SPIE, 01 2012.

[9] Peg Shippert. Why use hyperspectral imagery?. *Photogrammetric Engineering and Remote Sensing*, 70, 04 2004.

[10] C.-I Chang. *Hyperspectral imaging: Techniques for spectral detection and classification*. Kluwer Academic/Plenum, 2003.

[11] Karbhari Kale, Mahesh Solankar, Dhananjay Nalawade, Rajesh Dhumal, and Gite Hanumant. A research review on hyperspectral data processing and analysis algorithms. *Proceedings of the National Academy of Sciences, India Section A: Physical Sciences*, 87, 11 2017.

[12] Maider Vidal and Jose Amigo. Pre-processing of hyperspectral images. Essential steps before image analysis. *Chemometrics and Intelligent Laboratory Systems*, 117:138–148, 08 2012.

[13] Guolan Lu and Baowei Fei. Medical hyperspectral imaging: A review. *Journal of biomedical optics*, 19:10901, 01 2014.

[14] Subash K and Thyagharajan K K. Hyperspectral image compression algorithms – a review. *Advances in Intelligent Systems and Computing*, 325:127–138, 01 2014.

[15] Antonio Plaza, David Valencia, Javier Plaza, Juan Sanchez-Testal, S. Muñoz, and Soraya Blazquez. Parallel implementation of hyperspectral image processing algorithms. pages 940–943, 07 2006.

[16] Antonio Plaza, Jon Benediktsson, Jodie Boardman, Jason Brazile, Lorenzo Bruzzone, Gustau Camps-Valls, Jocelyn Chanussot, Mathieu Fauvel, Paolo Gamba, Anthony Gualtieri, Mattia Marconcini, James Tilton, and Giovanna Trianni. Recent advances in techniques for hyperspectral image processing. *Remote Sensing of Environment*, 113, 09 2009.

[17] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David Kaeli. Summarizing CPU and GPU design trends with product data, 2019.

[18] Barbara Chapman, Gabriele Jost and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming.* 2007.

[19] Jose Enrique Roman, Jose Miguel Alonso, Ignacio Blanquer, David Guerrero, Jacinto Javier Ibañez, Enrique Ramos, and Fernando Alvarruiz. *Ejercicios de programación paralela con OpenMP y MPI.* Editorial Universitat Politècnica de València, 2018.

[20] David R. Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. *Heterogeneous computing with OpenCL 2.0.* Morgan Kaufmann, 2015.

[21] Janusz Kowalik and Tadeusz Puzniakowski. *Using OpenCL: Programming massively parallel computers.* IOS Press, 2012.

[22] Ravishekhar Banger and Koushik Bhattacharyya. *OpenCL programming by example.* Packt Publishing, 2013.

[23] Shane Cook. CUDA programming : A developer's guide to parallel computing with GPUs, 2012.

[24] Sergio Bernabe, Carlos Garcia, Francisco Igual, Guillermo Botella, Manuel Prieto-Matias, and Antonio Plaza. Portability study of an OpenCL algorithm for automatic target detection in hyperspectral images. *IEEE Transactions on Geoscience and Remote Sensing*, 57(11):9499–9511, 08 2019.

[25] Maria Barreda, Sergio Barrachina, Sandra Catalan, Manuel F. Dolz, G. Fabregat, Rafael Mayo, and Enrique S. Quintana-Orti. An integrated framework for power-performance analysis of parallel scientific workloads. In *ENERGY 2013: The Third International*

Conference on Smart Grids, Green Communications and IT Energy-aware Technologies, 01 2013.

[26] Sergio Bernabe, Sebastian Lopez, Antonio Plaza, and Roberto Sarmiento. GPU implementation of an automatic target detection and classification algorithm for hyperspectral image analysis. *IEEE Geoscience and Remote Sensing Letters*, 10:221–225, 03 2013.

Apéndice A

Introduction

A.1. Motivation

Mankind has been able to improve its knowledge thanks to different new techniques that have been developed to observe the universe. Highly complex areas of expertise are being investigated, requiring advanced and efficient technologies to appreciate details that cannot otherwise be noticed. Among other remote sensing techniques, hyperspectral images have recently been used to study Earth's surface from space and aircraft [3] without the need for field experiments.

In remote sensing, hyperspectral images are an upgrade over traditional digital imagery, which only captures visible light within the electromagnetic spectrum (red, green and blue wavelengths). In contrast, hyperspectral imagery measures a broadest range of electromagnetic radiation, including infrared and ultraviolet wavelengths. The information provided by hyperspectral images can be used to identify materials present in a surface by solving the spectral unmixing problem, which is the principal challenge of this type of images.

Many activities can benefit from the use of hyperspectral imaging, such as identifying minerals for mining purposes, target detection for military purposes and all sorts of en-

vironmental studies. Detecting changes and damages to natural ecosystems, studying the effects of climate change and global warming or monitoring wildfires and other environmental disasters are only a few examples for the applications of hyperspectral imaging to those environmental studies. In addition, in recent years this remote sensing technique has also been implemented and put to use in industrial activities such as food quality control [4].

The size of hiperspectral images is a big challenge for processing all the information that they contain. Because of that, a big portion of all the data captured by hyperspectral sensors are stored unused in databases. On top of that, many of the applications previously mentioned require real-time processing, so an important goal is to be able to process images onboard the same platforms that hold the sensor. Improving existing algorithms to achieve real-time processing would speed up the process and reduce storing problems [5].

To solve these issues, we propose to accelerate one of the steps of processing hyperspectral images using parallel computing. Specifically, we used CUDA, OpenCL and OpenMP parallel programming models, testing their performance on multicore CPUs and GPUs (Graphics Processing Unit). In this paper, we focused in exploiting these technologies to help achieve real-time processing as efficiently as possible.

A.2. Objectives

The main goal of this project is to implement different accelerated versions of the FastSepNMF (Fast and Robust Recursive Algorithms for Separable Nonnegative Matrix Factorization) [6] algorithm using various parallel programming lenguajes. This algorithm can be used to process one of the three steps needed to calculate how materials are distributed in a hiperspectral image.

We have applied shared memory multiprocessing using OpenMP and heterogeneous programming using OpenCL and CUDA to the FastSepNMF algorithm. After discussing the

process of implementing the algorithm using all three languages, we will compare the performance and efficiency of all versions when executed in a test bench machine.

This goal will be achieved by completing several objectives listed below:

- Discussion of concepts needed to understand this specific area of knowledge. Definition of hyperspectral image, endmember and the spectral unmixing process.
- Analysis of the FastSepNMF algorithm.
- Translation of FastSepNMF to C language. Optimization of the C sequential version of the algorithm, allowing auto vectorization.
- Investigation of all languages that will be used to accelerate the algorithm.
- Implementation of the different accelerated versions of the algorithm.
- Discussion of obtained results, comparing performance, efficiency and quality of results from all versions of the algorithm.

A.3. Work plan

Figure A.1 shows a Gantt diagram illustrating all the different tasks that we have carried out during this project. Purple denotes tasks as they were originally planned. Orange denotes tasks that have been finished after the deadline or that have been resumed out of schedule, usually to improve a specific algorithm version after gaining extra knowledge in some area. Two patterns are shown in both colors: striped squares mean that the task was active during the whole period whilst plain squares mean that the task was considered finished at some point in that period.

Figura A.1: *Planning and contingencies of the work plan.*

A.4. Organization of this paper

Having defined the milestones in which the general objective has been organized, we are now concerned with specifying the chapters in which this paper has been structured, which are briefly detail below:

- **Hyperspectral analysis**: In this first chapter, basic concepts about hyperspectral imaging are discussed. Hyperspectral images and their spectral unmixing process (which takes care of the endmember extraction) are defined here. The need of applying paralell computing to the spectral unmixing problem is also presented here.

- **Implementation**: After presenting the context in which the project is developed, this chapter focuses in the FastSepNMF algorithm. FastSepNMF's characteristics are described here, as well as all the optimizations made during the translation to the C language. OpenMP, OpenCL and CUDA are introduced and the implementation of all versions of the algorithm using those languages are detailed in this chapter. The code of all developed versions can be found in GitHub[1].

- **Results**: Once the development section has been completed, the obtained results are discussed in this chapter. The quality of the algorithm's results is studied, as well as the performance and efficiencies of all implemented versions. Since both the different images and hardware used during the tests affects the results, the characteristics of those two elements are described in the results chapter.

- **Conclusions and future work**: Finally, the conclusions drawn from the results will be presented, determining whether the objective of this project has been met. In addition, possible lines of future research in relation to this project will be proposed.

[1] https://github.com/savary1/FastSepNMF_parallelization

Apéndice B

Conclusions and future work

B.1. Conclusions

This paper focuses on the study and exploitation of the parallelization potential of the FastSepNMF algorithm. This algorithm, which is based on the linear mixing model, is capable of extracting endmembers from a hyperspectral image with high noise tolerance and low computational cost by assuming the presence of pure pixels in the image. According to the article published at the end of 2013 [6] by FastSepNMF authors, no other similar algorithm offers such advantages.

Several versions of FastSepNMF have been implemented using different parallel programming paradigms to conduct this study. Shared memory multiprocessing has been applied using OpenMP, as well as heterogeneous programming using OpenCL and CUDA. Both the sequential version that has been implemented as a baseline and all the parallel versions have been optimized to take advantage of the hardware present in the testing machine. The testing machine was equipped with two Intel Xeon E5-2695 v3 CPUs, one Nvidia GeForce GTX 980 and one Nvidia GeForce GTX 1080.

The experimental tests were conducted using two different hyperspectral images. The

first one, Cuprite, was captured over Nevada, USA and represents a well-studied case. The second one is a large synthetic image that has been used to check the scalability of the different implemented versions of FastSepNMF on bigger images.

Both OpenCL and CUDA based versions of FastSepNMF achieve major performance improvements when using a GPU to accelerate the execution of the algorithm. OpenCL-GPU achieves almost 9 times better performance than the sequential version, whereas CUDA achieves a speedup of almost 12x when processing the synthetic image. OpenCL-CPU and OpenMP results are not as impressive, obtaining a speedup of only around 3x. All accelerated versions are well below the real-time processing threshold for the Cuprite image.

The most energy-efficient accelerated versions are, again, CUDA and OpenCL-GPU, in this exact order. In third and fourth place come the OpenMP and OpenCL-CPU versions. These are the expected results, as GPUs are much more efficient than CPUs in highly parallelistic scenarios.

This study concludes that the CUDA version of FastSepNMF is the most efficient. However, the OpenCL version achieves similar efficiencies when accelerated by a GPU and has the advantage of supporting many more non Nvidia GPUs, in addition to multicore CPUs and other kinds of hardware accelerators.

B.2. Lines of future work

To further investigate the FastSepNMF algorithm, we suggest testing the performance of all implemented versions on hardware different from the one used in the heterogeneous HPC system, and studying its scalability on larger real images. The reason behind the variable execution times could be unveiled by testing the OpenMP and OpenCL-CPU versions of FastSepNMF in processors other than the two Intel Xeons.

Another interesting investigation would be to check the performance and efficiency of the

OpenCL-GPU and CUDA versions of FastSepNMF using less powerful, but more efficient processors than the Intel Xeons present in the testing machine. These Intel Xeons have large parallel computing capabilities, which are not necessary when accelerating the algorithm with a GPU. Finding an optimal hardware configuration is mandatory before applying the algorithm to any real-world situation.

Finally, the use of the algorithm in controlled environments such as industrial quality control could be investigated. In these scenarios, it is more likely that there is at least one pure pixel for each endmember to be extracted. In such cases, it would be convenient to execute the algorithm on more compact hardware that could easily be transported next to the sensor, such as an FPGA (using the OpenCL version) or a SOC (System On Chip) similar to those used in smartphones.

Apéndice C

Reparto de trabajo

C.1. Aroa Ayuso Muñoz

A finales del curso anterior, tanto David como yo estábamos interesados en el trabajo que proponían Sergio y Guillermo, así que decidimos escribirles para solicitar algo más de información. En la primera reunión que tuvimos nos hablaron brevemente sobre el problema a tratar en el proyecto, además, nos compartieron varios trabajos similares sobre procesamiento de imágenes hiperespectrales que habían realizado compañeros en años anteriores. Durante el verano pudimos leernos estos trabajos para tener un conocimiento base sobre el tema.

Para familiarizarnos con los lenguajes que debíamos usar y comprender mejor las transformaciones por las que algoritmo pasaría, los tutores nos facilitaron un algoritmo de ejemplo, con el mismo propósito al que implementaríamos, en MATLAB, con sus traducciones a C y OpenCL. Junto con mi compañero, en primer lugar, estudiamos el algoritmo en MATLAB, para ello investigamos sobre las funciones usadas, hicimos pruebas sobre el código y anotamos las salidas y valores que se iban obteniendo. Después, una vez entendido lo suficiente el código, decidimos pasar a la traducción en C, en la que íbamos comparando

e identificando las transformaciones que se habían realizado. Para terminar con esta parte miramos brevemente la traducción a OpenCL. Mientras tanto empecé a leer los apuntes que mi compañero me había prestado sobre los paradigmas de programación paralela, ya que hasta el momento no había tenido ningún contacto con ellos.

Una vez terminado el análisis, los tutores nos facilitaron el algoritmo con el que trabajaríamos a partir de ese momento, FastSepNMF. Dado que ambos trabajaríamos sobre el mismo algoritmo, las primeras partes las hicimos conjuntamente. Primero, estudiamos el algoritmo en MATLAB y comentamos cada línea de la implementación de la forma más descriptiva posible. Para ello investigamos sobre las funciones propias del lenguaje y anotamos todas las transformaciones y rotaciones que se hacían sobre las matrices y vectores. Posteriormente, implementamos la primera versión en C realizando una traducción exacta. Estas dos partes fueron las que más tiempo no llevó realizar debido a la falta de manejo con matrices, lo que nos dificultó entender la orientación de la imagen y cómo debía ser leída y guardada.

Después de obtener la primera versión, David se dio cuenta que sería más eficiente si realizábamos las lecturas sobre la imagen de manera consecutiva, por lo que yo me centré en modificar la forma en la que guardar los datos de la imagen mientras que él se encargó de simplificar y eliminar las operaciones innecesarias. Con ese fin, modifiqué el código para realizar la escritura de la imagen por bandas en vez de por imágenes, cómo se había implementado al principio, y adapté el resto de las lecturas. Además, probé cambiando el tipo de lo datos, comprobando si reduciendo la precisión se obtenían mejores tiempos sin repercutir negativamente en el resultado.

En una de las conversaciones con los tutores nos comentaron que podríamos hacer uso de instrucciones vectoriales o bibliotecas matemáticas para optimizar las partes del algoritmo que eran más costosas. Tras un tiempo investigando encontramos la biblioteca MKL (Intel Math Kernel Library), de la que me encargué de estudiar y aplicar al algoritmo. Encontré

varias funciones matemáticas vectoriales y matriciales que podrían sustituir al bucle que más tiempo consumía. Después de las pruebas vi que los tiempos obtenidos no mejoraban así que decidimos no añadir estas funciones a la implementación final.

Al finalizar la implementación y modificación de la versión en serie, comenzamos las implementaciones de OpenCL y CUDA, David se encargó de la primera mientras que yo de la segunda. Esta fue una de las tareas con mayor dificultad para mí, ya que nunca había trabajado con un lenguaje similar. En primer lugar, volví a revisar lo apuntes de David sobre CUDA e investigué en las guías de desarrollo y en varios foros. Después, procedí a implementar las regiones paralelas de la versión OpenMP, una vez obtenido el primer *kernel* y las funciones principales para su lanzamiento el resto no supusieron problema. Al tener un nuevo esquema de ejecución modifiqué y eliminé la copia de vectores entre el *host* y el *device* que ya no era necesaria, además, investigué una forma alternativa para hacer la reserva de memoria de algunos vectores con el fin de reducir el tiempo de transferencia, obteniendo así la versión final.

Finalmente, empezamos a escribir la memoria haciendo uso de LaTeX. De manera conjunta realizamos las mediciones de los tiempos y consumos para poder efectuar los cálculos expuestos en el apartado de resultados experimentales. Repartiendo los apartados y leyendo y corrigiendo lo que había escrito el otro, terminamos la memoria.

C.2. David Savary Martínez

El trabajo comenzó en Julio de 2019, cuando Sergio y Guillermo nos presentaron la idea general del proyecto en la primera reunión que tuvimos con ellos. Al encontrar el tema interesante, nos facilitaron material relacionado con el procesamiento de imágenes hiperespectrales, que leímos durante el verano para introducirnos en el campo que trata este trabajo de fin de grado.

Más tarde, al comenzar el curso, nuestra primera tarea fue analizar el código, tanto en MATLAB como en C, de otro algoritmo de extracción de *endmembers* completamente diferente al tratado en este trabajo. En este análisis, que realizamos de forma conjunta mi compañera Aroa y yo, fuimos relacionando cada una de las operaciones que se realizaban en MATLAB con el código correspondiente en C. Este proceso fue largo debido a que no contaba con ningún tipo de experiencia con el lenguaje ni el paradigma de programación de MATLAB. También estudiamos una implementación en OpenCL de este algoritmo, aunque de forma menos exhaustiva, ya que este paso depende en gran medida de las partes que se pueden acelerar de cada algoritmo y, al menos en mi caso, ya estaba familiarizado con OpenCL.

Una vez elegido el algoritmo que íbamos a acelerar (FastSepNMF), procedimos a analizar su implementación en MATLAB, de nuevo de forma conjunta. Éste análisis fue aún más detallado que el anterior, ya que investigamos el funcionamiento de todas las funciones utilizadas en la implementación original y anotamos cada uno de los cambios que se realizan a las matrices durante una ejecución del algoritmo. A partir del pseudocódigo generado conseguimos la primera versión funcional del algoritmo en C, tras varias semanas escribiendo y depurando el código.

Realizando un *profiling* a la primera traducción del algoritmo, llegué a la conclusión de que los principales cuellos de botella eran tanto los accesos no adyacentes a memoria al

operar con las matrices, como los bucles que se encargaban de la inicialización del algoritmo y de uno de los pasos que se realiza después de seleccionar cada *endmember* (como se detalla en el Capítulo 3). En este momento nos repartimos las tareas: mi compañera se encargó de optimizar los accesos a memoria, mientras yo modifiqué el código para evitar operaciones innecesarias, simplifiqué los bucles que más tarde se paralelizaron para facilitar la implementación de las siguientes versiones y modifiqué los bucles que el compilador ICC no era capaz de vectorizar para permitir que finalmente sí pudieran autovectorizarse.

Tras combinar las optimizaciones que ambos realizamos en una única versión secuencial del algoritmo, investigué las opciones que GCC e ICC ofrecen para aumentar el rendimiento del código. Siguiendo las guías de ambos compiladores y con la ayuda de libros de programación de OpenMP, conseguí reducir de forma considerable los tiempos de ejecución, obteniendo un rendimiento prácticamente idéntico compilando con GCC e ICC.

Para finalizar la etapa de desarrollo de este trabajo, quedaba por implementar las versiones OpenCL y CUDA. Yo me encargué del desarrollo de la versión OpenCL, que fue un gran desafío a pesar de tener experiencia previa con esta tecnología. La complejidad del código y el tamaño de los datos resultaron en una gran cantidad de tiempo invertido en depurar, usando GDB, la implementación de las regiones paralelas de OpenMP en *kernels* OpenCL. También investigué y experimenté con todas las opciones que ofrece OpenCL para trasferir datos entre el *host* y el *device*, con el objetivo de conseguir la máxima eficiencia al ejecutar el código tanto en una CPU *multicore* como en una GPU. Para finalizar esta implementación de la versión OpenCL, escribí e incorporé los *kernels* de reducción al código, lo que no llevó tanto tiempo como el desarrollo de los primeros *kernels* debido a todo el nuevo conocimiento adquirido sobre OpenCL.

El paso final consistió en la redacción de esta memoria. Para ello fue necesario realizar todas las pruebas de rendimiento y consumo a las versiones desarrolladas durante los meses anteriores. Aroa y yo colaboramos en el proceso de toma de tiempos y en las mediciones

de los consumos, para las cuales fue necesario modificar todas las versiones analizadas para hacer uso del framework Pmlib. Tras analizar los datos obtenidos y documentarnos para ser capaces de valorarlos, procedimos a redactar la memoria. Durante la redacción, cada uno se encargó de partes concretas, revisando y modificando de forma conjunta la totalidad del documento para perfeccionar todos los apartados que pudieran ser mejorados.